Tài liệu PostgreSQL 9.0.13

Nhóm phát triển toàn cầu PostgreSQL

V. Lập trình máy chủ

Dịch sang tiếng Việt: Lê Trung Nghĩa, letrungnghia.foss@gmail.com

Dịch xong: 16/01/2015

Bản gốc tiếng Anh:

http://www.postgresql.org/files/documentation/pdf/9.0/postgresql-9.0-A4.pdf

---

PostgreSQL 9.0.13 Documentation

The PostgreSQL Global Development Group

V. Server Programming

Lời người dịch

Tài liệu PostgreSQL 9.0.13 của nhóm phát triển toàn cầu PostgreSQL xuất bản năm 2013, như trong phần LỜI NÓI ĐẦU ở bên dưới giới thiệu, gồm 7 phần chính, đánh số từ I tới VII và phần dành cho các phụ lục, được đánh số VIII, tổng cộng dài 2.364 trang.

Vào tháng 03/2014, hai phần đầu của tài liệu này đã được dịch xong, gồm:

- Phần I: Sách chỉ dẫn, là một giới thiệu không chính thức cho những người mới sử dụng.
- Phần II: Ngôn ngữ SQL, viết về môi trường ngôn ngữ truy vấn SQL, bao gồm cả các dạng và các hàm dữ liệu, cũng như việc tinh chỉnh hiệu năng ở mức của người sử dụng. Mọi người sử dụng PostgreSQL đều nên đọc phần này.

Bản dịch sang tiếng Việt của Phần I và Phần II hiện có thể tải về từ địa chỉ:

https://www.dropbox.com/s/t5sj1z55ifjxeu8/postgresql-9.0-A4-Part-I-II-Vi-13032014.pdf

Tiếp đến, vào tháng 11/2014, hai phần tiếp sau của tài liệu này đã được dịch xong, gồm:

- Phần III mô tả sự cài đặt và quản trị máy chủ. Từng người mà quản lý một máy chủ PostgreSQL, dù là để sử dụng riêng hay vì những lý do khác, nên đọc phần này.
- Phần IV mô tả các giao diện lập trình cho các chương trình máy trạm PostgreSQL.

Bản dịch sang tiếng Việt của Phần III và Phần IV hiện có thể tải về từ địa chỉ:

https://www.dropbox.com/s/ay8lxqmfzug5yl3/postgresql-9.0-A4-Part-III-IV-Vi-12112014.pdf?dl=0

Bản dịch lần này là phần tiếp theo của 4 phần trên cũng đã được dịch xong trước và đưa ra để các độc giả sử dụng sớm, đó là:

- Phần V bao gồm các thông tin cho những người sử dụng cao cấp về các khả năng mở rộng của máy chủ. Các chủ đề bao gồm các dạng và hàm dữ liệu do người sử dụng định nghĩa.

Chính vì tài liệu không được dịch xong hoàn toàn tất cả các phần cùng một lúc, nên trong quá trình sử dụng, có một số nội dung tham chiếu ở các phần từ Phần I đến Phần V tới các phần còn lại của tài liệu các độc giả chỉ có thể xem nội dung ở bản gốc tiếng Anh (như theo đường dẫn ở bìa trước của tài liệu) mà chưa có phần dịch sang tiếng Việt. Rất mong được các độc giả thông cảm.

Hy vọng các phần tiếp sau sẽ được dịch sang tiếng Việt sớm.

Việc dịch thuật chắc chắn không khỏi có những lỗi nhất định, rất mong các bạn độc giả, một khi phát hiện được, xin liên lạc với người dịch và đóng góp ý kiến về cách chỉnh sửa để tài liệu sẽ ngày một có chất lượng tốt hơn, phục vụ cho các bạn độc giả và mọi người có nhu cầu được tốt hơn. Xin chân thành cảm ơn trước các bạn độc giả về các đóng góp chỉnh sửa đó.

Mọi thông tin đóng góp chỉnh sửa cho bản dịch tiếng Việt, xin vui lòng gửi vào địa chỉ thư điện tử:

letrungnghia.foss@gmail.com

Chúc các độc giả thành công!

Hà Nội, ngày 16/01/2015

Lê Trung Nghĩa

Mục lục

Lời nói đầu

Cuốn sách này là tài liệu chính thức của PostgreSQL. Nó đã được các lập trình viên và những người tình nguyện khác của PostgreSQL viết song song với sự phát triển của phần mềm PostgreSQL. Nó mô tả tất cả các chức năng mà phiên bản hiện hành của PostgreSQL chính thức hỗ trợ.

Để làm cho số lượng lớn các thông tin về PostgreSQL có khả năng quản lý được, cuốn sách này đã được tổ chức trong vài phần. Mỗi phần có mục đích cho từng lớp người sử dụng khác nhau, hoặc cho các giai đoạn khác nhau của người sử dụng đối với kinh nghiệm về PostgreSQL của họ:

- Phần I là một giới thiệu không chính thức cho những người mới sử dụng.
- Phần II viết về môi trường ngôn ngữ truy vấn SQL, bao gồm cả các dạng và các hàm dữ liệu, cũng như việc tinh chỉnh hiệu năng ở mức của người sử dụng. Mọi người sử dụng PostgreSQL đều nên đọc phần này.
- Phần III mô tả sự cài đặt và quản trị máy chủ. Từng người mà quản lý một máy chủ PostgreSQL, dù là để sử dụng riêng hay vì những lý do khác, nên đọc phần này.
- Phần IV mô tả các giao diện lập trình cho các chương trình máy trạm PostgreSQL.
- Phần V bao gồm các thông tin cho những người sử dụng cao cấp về các khả năng mở rộng của máy chủ. Các chủ đề bao gồm các dạng và hàm dữ liệu do người sử dụng định nghĩa.
- Phần VI bao gồm các thông tin tham chiếu về các lệnh SQL, các chương trình máy trạm và máy chủ. Phần này hỗ trợ các phần khác với các thông tin được sắp xếp theo lệnh hoặc chương trình.
- Phần VII bao gồm các thông tin hỗn hợp có thể được sử dụng cho các lập trình viên của PostgreSQL.

V. Lập trình máy chủ

Phần này là về việc mở rộng chức năng máy chủ với các hàm, các dạng dữ liệu, các trigger, ... do người sử dụng định nghĩa. Chúng là các chủ đề cao cấp có lẽ sẽ được tiếp cận chỉ sau khi tất cả các tài liệu người sử dụng khác về PostgreSQL đã được hiểu. Các chương tiếp sau trong phần này mô tả các ngôn ngữ lập trình phía máy chủ, có sẵn trong phân phối PostgreSQL cũng như các vấn đề có liên quan tới các ngôn ngữ lập trình phía máy chủ. Là cơ bản để đọc ít nhất các phần đầu của Chương 35 (đề cập tới các hàm) trước khi đi sâu vào tư liệu các ngôn ngữ lập trình phía máy chủ.

Chương 35. Mở rộng SQL

Trong các phần sau đây, chúng tôi sẽ thảo luận cách mà bạn có thể mở rộng ngôn ngữ truy vấn SQL của PostgreSQL bằng việc thêm:

- các hàm (bắt đầu ở Phần 35.3)
- tổng hợp (bắt đầu ở Phần 35.10)
- các dạng dữ liệu (bắt đầu ở Phần 35.11)
- các toán tử (bắt đầu ở Phần 35.12)
- các lớp toán tử cho các chỉ số (bắt đầu ở Phần 35.14)

35.1. Sự mở rộng làm việc như thế nào

PostgreSQL là có khả năng mở rộng được vì sự vận hành của nó là hướng catalog. Nếu bạn quen với các hệ thống cơ sở dữ liệu quan hệ tiêu chuẩn, thì bạn biết rằng chúng lưu trữ thông tin về các cơ sở dữ liệu, các bảng, các cột, ..., trong những gì thường được biết như là các catalog hệ thống. (Một số hệ thống gọi điều này là thư mục dữ liệu). Các catalog đó hiển thị cho người sử dụng như các bảng giống như bất kỳ catalog nào khác, nhưng hệ quản trị cơ sở dữ liệu (DBMS) lưu trữ việc kế toán nội bộ của nó trong chúng. Sự khác biệt chính giữa PostgreSQL và các hệ thống cơ sở dữ liệu quan hệ tiêu chuẩn là PostgreSQL lưu trữ nhiều thông tin hơn nhiều trong các catalog của nó: không chỉ thông tin về các bảng và các cột, mà còn cả thông tin về các dạng dữ liệu, các hàm, các phương pháp truy cập trong các bảng đó, và hơn thế. Các bảng đó có thể được người sử dụng tùy biến, và vì PostgreSQL dựa sự vận hành của nó vào các bảng đó, điều này có nghĩa là PostgreSQL có thể được người sử dụng mở rộng. Để so sánh, các hệ thống cơ sở dữ liệu thông thường chỉ có thể được mở rộng bằng việc thay đổi các thủ tục được viết cứng (hardcode) vào trong mã nguồn bằng việc tải các module được nhà bán hàng DBMS viết một cách đặc biệt.

Hơn nữa, máy chủ PostgreSQL có thể kết hợp mã do người sử dụng viết vào bản thân nó thông qua việc tải động. Đó là, người sử dụng có thể chỉ định một tệp mã đối tượng (như, một thư viện chia sẻ) mà triển khai một dạng (type) hoặc hàm mới, và PostgreSQL sẽ tải nó như được yêu cầu. Mã được viết trong SQL thậm chí là thông thường hơn để thêm vào máy chủ. Khả năng để tùy biến này sự vận hành của nó "ngay tại trận" làm cho PostgreSQL là duy nhất phù hợp cho việc tạo bản mẫu nhanh các ứng dụng và các cấu trúc lưu trữ mới.

35.2. Hệ thống các dạng của PostgreSQL

Các dạng dữ liệu của PostgreSQL được chia thành 2 dạng cơ bản, tạo thành các dạng, các miền, và các dạng giả (pseudo-type).

35.2.1. Dạng cơ bản

Các dạng cơ bản là các dạng, giống như int4, được triển khai bên dưới mức ngôn ngữ SQL (thường

ở một ngôn ngữ mức thấp như C). Chúng thường tương xứng với những gì thường được biết tới như là các dạng dữ liệu trừu tượng. PostgreSQL chỉ có thể vận hành trong các dạng như vậy thông qua các hàm được người sử dụng cung cấp và chỉ hiểu hành vi của các dạng như vậy ở mức độ mà người sử dụng mô tả chúng. Các dạng cơ bản được chia tiếp thành các dạng vô hướng và mảng. Đối với từng dạng vô hướng, một dạng mảng tương ứng tự động được tạo ra mà có thể nắm giữ các mảng có kích cỡ khác nhau của dạng vô hướng đó.

35.2.2. Dạng tổng hợp

Các dạng tổng hợp, hoặc các dạng hàng, được tạo ra bất kỳ khi nào người sử dụng tạo một bảng. Cũng là có khả năng sử dụng CREATE TYPE để xác định một dạng tổng hợp "đứng một mình" (stand-alone) với không bảng có liên quan nào. Một dạng tổng hợp đơn giản là một danh sách các dạng với các tên trường có liên quan. Giá trị của một dạng tổng hợp là một hàng hoặc một bản ghi các giá trị trường. Người sử dụng có thể truy cập các trường thành phần từ các truy vấn SQL. Hãy tham chiếu tới Phần 8.15 để có thêm thông tin về các dạng tổng hợp.

35.2.3. Miền

Miền được dựa vào dạng cơ bản đặc biệt và vì nhiều mục đích là trao đổi được với dạng cơ bản của nó. Tuy nhiên, một miền có thể có các ràng buộc hạn chế các giá trị hợp lệ của nó tới một tập con của những gì dạng cơ bản nằm bên dưới có thể cho phép.

Các miền có thể được tạo ra bằng việc sử dụng lệnh SQL CREATE DOMAIN. Sự tạo ra và sử dụng chúng không được thảo luận trong chương này.

35.2.4. Dạng giả

Có một ít "các dạng giả" (pseudo type) cho các mục đích đặc biệt. Các dạng giả không thể xuất hiện như các cột của các bảng hoặc các thuộc tính của các dạng tổng hợp, nhưng chúng có thể được sử dụng để khai báo biến và các dạng kết quả của các hàm. Điều này đưa ra một cơ chế bên trong hệ thống dạng để nhận diện các lớp hàm đặc biệt.

35.2.5. Dạng đa hình (Polymorphic type)

4 dạng giả có sự quan tâm đặc biệt là anyelement, anyarray, anynonarray, và anyenum, chúng được gọi là các *hàm đa hình (polymorphic)*. Bất kỳ hàm nào được khai báo bằng việc sử dụng các dạng đó sẽ là một *hàm đa hình*. Một hàm đa hình có thể vận hành trong nhiều dạng dữ liệu khác nhau, với (các) dạng dữ liệu đặc thù đang được các dạng dữ liệu xác định thực sự được truyền tới nó trong một lời gọi đặc biệt.

Các đối số và các kết quả đa hình bị ràng buộc với nhau và sẽ được đề cập tới một dạng dữ liệu đặc biệt khi một truy vấn gọi một hàm đa hình được phân tích cú pháp. Mỗi vị trí (hoặc đối số hoặc giá trị trả về) được khai báo như là anyelement được cho phép để có bất kỳ dạng dữ liệu thực sự đặc biệt nào, nhưng trong bất kỳ lời gọi được đưa ra nào chúng cũng phải tất cả là dạng thực sự y hệt nhau. Từng vị trí được khai báo như là anyarray có thể có bất kỳ dạng dữ liệu mảng nào, nhưng tương tự chúng tất cả phải là dạng y hệt nhau. Nếu có các vị trí được khai báo là anyarray và các vị trí khác được khai báo là anyelement, nhưng thêm ràng buộc bổ sung thì dạng thực tế đó phải không

là một dạng mảng. anyenum được đối xử chính xác y hệt như anyelement, nhưng thêm ràng buộc bổ sung mà dạng thực tế đó phải là một dạng đánh số.

Vì thế, khi nhiều hơn một vị trí đối số được khai báo với một dạng đa hình, thì ảnh hưởng thực là chỉ các tổ hợp nhất định của các dạng đối số thực được cho phép. Ví dụ, một hàm được khai báo như là equal(anyelement, anyelement) sẽ lấy bất kỳ 2 giá trị đầu vào nào, miễn là chúng sẽ là dạng dữ liệu y hệt nhau.

Khi giá trị trả về của một hàm được khai báo như một dạng đa hình, sẽ phải có ít nhất một vị trí đối số cũng là đa hình, và dạng dữ liệu thực sự được cung cấp như là đối số xác định dạng kết quả thực cho lời gọi đó. Ví dụ, nếu đã không có một cơ chế đánh chỉ số dưới nào của mảng, thì cơ chế có thể xác định hàm triển khai việc đánh chỉ số dưới như subscript(anyarray, integer) returns anyelement. Sự khai báo này ràng buộc đối số đầu tiên thực sự sẽ là một dạng mảng, và cho phép trình phân tích cú pháp phỏng đoán dạng kết quả đúng từ dạng đối số thực đầu tiên. Ví dụ khác là hàm được khai báo như là f(anyarray) returns anyenum sẽ chỉ chấp nhận các mảng dạng đánh số.

Lưu ý rằng anynonarray và anyenum không đại diện cho các biến dạng tách biệt; chúng là dạng y hệt như anyelement, hệt với ràng buộc bổ sung. Ví dụ, việc khai báo hàm như là f(anyelement, anyenum) là tương đương với việc khai báo nó như là f(anyenum, anyenum): cả 2 đối số thực sẽ có cùng y hệt dạng đánh số.

Một hàm bất định (hàm có số các biến khác nhau, như trong Phần 35.4.5) có thể là đa hình: điều này được hoàn tất bằng việc khai báo tham số cuối cùng của nó như là VARIADIC anyarray. Vì các mục đích đối số trùng khớp và xác định dạng kết quả thực, một hàm như vậy hành xử y hệt nếu bạn đã viết số các tham số anynonarray đúng phù hợp.

35.3. Hàm do người sử dụng định nghĩa

PostgreSQL đưa ra 4 dạng hàm:

- các hàm ngôn ngữ truy vấn (các hàm được viết trong SQL) (Phần 35.4)
- các hàm ngôn ngữ thủ tục (các hàm được viết trong, ví dụ, PL/pgSQL hoặc PL/Tcl) (Phần 35.7)
- các hàm nội bộ (Phần 35.8)
- các hàm ngôn ngữ C (Phần 35.9)

Mỗi dạng hàm có thể lấy các dạng cơ bản, các dạng tổng hợp, hoặc các tổng hợp của chúng như các đối số (các tham số). Hơn nữa, từng dạng hàm có thể trả về dạng cơ bản hoặc dạng tổng hợp. Các hàm cũng có thể được xác định để trả về các tập hợp cơ bản hoặc các giá trị tổng hợp.

Nhiều dạng hàm có thể lấy hoặc trả về các dạng giả nhất định (như các dạng đa hình), nhưng các tiện ích có sẵn là khác nhau. Hãy tư vấn mô tả của từng dạng hàm để có thêm chi tiết.

Là dễ dàng nhất để xác định các hàm SQL, sao cho chúng ta sẽ bắt đầu bằng việc thảo luận chúng. Hầu hết các khái niệm được trình bày cho các hàm SQL là mang tới các dạng hàm khác được.

Qua chương này, có thể là hữu ích để xem xét trang tham chiếu của lệnh CREATE FUNCTION để

hiểu các ví dụ đó tốt hơn. Vài ví dụ từ chương này có thể được thấy trong funcs.sql và funcs.c trong thư mục src/tutorial trong phân phối nguồn của PostgreSQL.

35.4. Hàm của ngôn ngữ truy vấn (SQL)

Các hàm SQL thực thi một danh sách tùy ý các lệnh SQL, trả về kết quả của truy vấn cuối cùng trong danh sách. Trong trường hợp đơn giản (không có tập hợp), thì hàng đầu của kết quả truy vấn cuối cùng sẽ được trả về. (Hãy nhớ trong đầu rằng "hàng đầu" của một kết quả nhiều hàng không được xác định tốt trừ phi bạn sử dụng ORDER BY). Nếu truy vấn cuối cùng bỗng nhiên không trả về hàng nào cả, thì giá trị null sẽ được trả về.

Như một sự lựa chọn, một hàm SQL có thể được khai báo để trả về một tập hợp, bằng việc chỉ định dạng trả về của hàm như là SETOF *sometype*, hoặc tương đương bằng việc khai báo nó như là RETURNS TABLE(*columns*). Trong trường hợp này, tất cả các hàng của kết quả truy vấn cuối cùng sẽ được trả về. Chi tiết hơn sẽ có ở bên dưới.

Thân của một hàm SQL phải là một danh sách các lệnh SQL được phân cách nhau bằng các dấu chấm phẩy. Một dấu chấm phẩy sau lệnh cuối cùng là tùy chọn. Trừ phi hàm đó được khai báo để trả về void, lệnh cuối cùng phải là một lệnh SELECT, hoặc một lệnh INSERT, UPDATE, hoặc DELETE mà có một mệnh đề RETURNING.

Bất kỳ bộ sưu tập các lệnh nào trong ngôn ngữ SQL cũng có thể được đóng gói cùng và được xác định như một hàm. Ngoài các truy vấn SELECT, các lệnh có thể bao gồm các truy vấn sửa dữ liệu (INSERT, UPDATE, và DELETE), cũng như các lệnh SQL khác. (Ngoại lệ duy nhất là bạn không thể đặt các lệnh BEGIN, COMMIT, ROLLBACK, hoặc SAVEPOINT vào một hàm SQL). Tuy nhiên, lệnh cuối cùng phải là SELECT hoặc có mệnh đề RETURNING trả về bất kỳ điều gì được chỉ định như là dạng trả về của hàm. Như một sự lựa chọn, nếu bạn muốn xác định một hàm SQL mà thực thi các hành động nhưng không có giá trị hữu ích nào trả về, thì bạn có thể xác định nó như là việc trả về void. Ví dụ, hàm này loại bỏ các hàng với các mức lương là âm từ bảng emp:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
        DELETE FROM emp
                WHERE salary < 0;
' LANGUAGE SQL;
SELECT clean_emp();

clean_emp
----------
  (1 row)
```

Cú pháp của lệnh CREATE FUNCTION đòi hỏi thân hàm phải được viết như một hằng chuỗi. Thường là thuận tiện nhất để sử dụng dấu đô la ($) (xem Phần 4.1.2.4) cho hằng chuỗi đó.

Nếu bạn chọn sử dụng cú pháp hằng chuỗi có các dấu ngoặc đơn thông thường, thì bạn phải đúp bản các dấu ngoặc đơn (') và các dấu chéo ngược (\ ) (giả thiết cú pháp chuỗi thoát) trong thân hàm (xem Phần 4.1.2.1).

Các đối số cho hàm SQL được tham chiếu trong thân hàm bằng việc sử dụng cú pháp $n : $1 tham chiếu tới đối số đầu tiên, $2 tới đối số thứ 2, và cứ thế. Nếu một đối số là dạng tổng hợp, thì dấu chấm (.), như $1.name, có thể được sử dụng để truy cập các thuộc tính của đối số đó. Các đối số chỉ

có thể được sử dụng như các giá trị dữ liệu, chứ không là các mã định danh. Vì thế ví dụ này là chấp nhận được:

```
INSERT INTO mytable VALUES ($1);
```

nhưng điều này thì sẽ không làm việc:

```
INSERT INTO $1 VALUES (42);
```

35.4.1. Hàm SQL ở dạng cơ bản

Hàm SQL có khả năng đơn giản nhất không có các đối số và đơn giản trả về một dạng cơ bản, như integer:

```
CREATE FUNCTION one() RETURNS integer AS $$
        SELECT 1 AS result;
$$ LANGUAGE SQL;
-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
        SELECT 1 AS result;
' LANGUAGE SQL;
SELECT one();

one
-----
  1
```

Lưu ý rằng chúng ta đã xác định một tên hiệu cột bên trong thân hàm cho kết quả của hàm đó (với tên result), nhưng tên hiệu cột này không nhìn thấy bên ngoài hàm đó. Vì thế, kết quả được gắn nhãn one thay vì result.

Hầu như là dễ dàng để xác định các hàm SQL mà lấy các dạng cơ bản như là các đối số. Trong ví dụ bên dưới, lưu ý cách chúng ta tham chiếu tới các đối số bên trong hàm như $1 và $2.

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
        SELECT $1 + $2;
$$ LANGUAGE SQL;
SELECT add_em(1, 2) AS answer;

answer
----------
       3
```

Đây là một hàm hữu dụng hơn, nó có thể được sử dụng để ghi nợ một tài khoản ngân hàng:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
        UPDATE bank
                SET balance = balance - $2
                WHERE accountno = $1;
        SELECT 1;
$$ LANGUAGE SQL;
```

Một người sử dụng có thể thực thi hàm này để ghi nợ cho tài khoản 17 bằng &100.00 như sau:

```
SELECT tf1(17, 100.0);
```

Trong thực tế người ta có thể thích một kết quả hữu dụng hơn từ hàm so với hằng số 1, nên một định nghĩa có khả năng hơn là:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
        UPDATE bank
```

```
                SET balance = balance - $2
                WHERE accountno = $1;
        SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

nó chỉnh bản quyết toán và trả về bản quyết toán mới. Thứ y hệt có thể được thực hiện trong 1 lệnh bằng việc sử dụng RETURNING:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
        UPDATE bank
                SET balance = balance - $2
                WHERE accountno = $1
        RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.2. Hàm SQL ở dạng tổng hợp

Khi viết các hàm với các đối số các dạng tổng hợp, chúng ta phải không chỉ chỉ định đối số nào chúng ta muốn (như chúng ta đã làm ở trên với $1 và $2) mà còn cả thuộc tính (trường) mong muốn của đối số đó. Ví dụ, giả sử emp là một bảng có dữ liệu các nhân viên, và vì thế cũng có tên dạng tổng hợp của từng hàng trong bảng đó. Đây là một hàm double_salary mà tính toán lương tháng của ai đó sẽ là bao nhiêu nếu nó từng là gấp đôi:

```
CREATE TABLE emp (
        name text,
        salary numeric,
        age integer,
        cubicle point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
        SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
        FROM emp
        WHERE emp.cubicle ~= point '(2,1)';

name | dream
--------+-------
Bill    | 8400
```

Lưu ý sử dụng cú pháp $1.salary để chọn một trường giá trị hàng của đối số. Cũng lưu ý cách gọi lệnh SELECT sử dụng * để chọn toàn bộ hàng hiện hành của một bảng như một giá trị tổng hợp. Hàng bảng đó có thể như một sự lựa chọn được tham chiếu bằng việc sử dụng chỉ tên bảng đó, như:

```
SELECT name, double_salary(emp) AS dream
        FROM emp
        WHERE emp.cubicle ~= point '(2,1)';
```

nhưng sử dụng này không được tán thành vì nó dễ gây bối rối.

Đôi khi là thuận tiện để xây dựng một giá trị đối số tổng hợp tại chỗ. Điều này có thể được thực hiện với sự xây dựng ROW. Ví dụ, chúng ta có thể tinh chỉnh dữ liệu đang được truyền tới hàm đó:

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
        FROM emp;
```

Cũng có khả năng để xây dựng một hàm trả về một dạng tổng hợp. Đây là một ví dụ về một hàm trả về một hàng emp duy nhất:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
        SELECT text 'None' AS name,
                1000.0 AS salary,
                25 AS age,
                point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

Trong ví dụ này chúng ta đã chỉ định từng trong số các thuộc tính với một giá trị hằng, nhưng bất kỳ tính toán nào cũng có thể bị thay thế cho các hàm đó.

Lưu ý 2 điều quan trọng về việc xác định hàm đó:

- Trật tự liệt kê được chọn trong truy vấn phải chính xác y hệt như trật tự theo đó các cột xuất hiện trong bảng có liên quan tới dạng tổng hợp đó. (Việc đặt tên các cột, như chúng ta đã nêu ở trên, là không phù hợp cho hệ thống).
- Bạn phải gõ ra các biểu thức để khớp với định nghĩa của dạng tổng hợp, hoặc bạn sẽ gặp các lỗi như thế này:

  ```
  ERROR: function declared to return emp returns varchar instead of text at column 1
  ```

Một cách khác để xác định hàm y hệt là:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
        SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Ở đây chúng ta đã viết một SELECT mà trả về chỉ một cột duy nhất của dạng tổng hợp đúng. Điều này thực sự không là tốt hơn trong tình huống này, nhưng đây là một lựa chọn thay thế có ích trong vài trường hợp - ví dụ, nếu chúng ta cần tính toán kết quả bằng việc gọi hàm khác mà trả về giá trị tổng hợp mong muốn.

Chúng ta có thể gọi hàm này trực tiếp theo 2 cách:

```
SELECT new_emp();

        new_emp
----------------------------------
(None,1000.0,25,"(2,2)")

SELECT * FROM new_emp();

name | salary  | age  | cubicle
--------+----------+------+-----------
None  | 1000.0 |  25  | (2,2)
```

Cách thứ 2 được mô tả đầy đủ hơn trong Phần 35.4.7.

Khi bạn sử dụng một hàm mà trả về một dạng tổng hợp, thì bạn có thể chỉ muốn một trường (thuộc tính) từ kết quả của nó. Bạn có thể làm điều đó với cú pháp như thế này:

```
SELECT (new_emp()).name;

name
--------
None
```

Các dấu ngoặc thừa là cần thiết để giữ cho trình phân tích cú pháp khỏi bị lẫn. Nếu bạn cố làm mà không có chúng, thì bạn đôi khi gặp thứ giống thế này:

```
SELECT new_emp().name;
ERROR: syntax error at or near "."
LINE 1: SELECT new_emp().name;
```

Một lựa chọn khác là sử dụng ký hiệu hàm cho việc trích xuất một thuộc tính. Cách đơn giản để giải thích điều này là chúng ta có thể sử dụng các ký hiệu attribute(table) và table.attribute lẫn nhau.

```
SELECT name(new_emp());

name
--------
None

-- Điều này là hệt như:
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30;

SELECT name(emp) AS youngster FROM emp WHERE age(emp) < 30;

youngster
--------------
   Sam
   Andy
```

> **Mẹo**: Sự tương đương giữa ký hiệu hàm và ký hiệu thuộc tính làm cho có khả năng sử dụng các hàm trong các dạng tổng hợp để mô phỏng "các trường được tính toán". Ví dụ, sử dụng định nghĩa trước đó cho double_salary(emp), chúng ta có thể viết
>
> ```
> SELECT emp.name, emp.double_salary FROM emp;
> ```
>
> Một ứng dụng sử dụng điều này sẽ không cần trực tiếp phải nhận thức được rằng double_salary không phải là một cột thực của bảng. (Bạn cũng có thể mô phỏng các trường tính toán bằng các kiểu nhìn - views).

Một cách khác để sử dụng một hàm trả về một dạng tổng hợp là truyền kết quả sang hàm khác mà chấp nhận dạng hàng đúng như đầu vào:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
        SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());

getname
-------------
  None
 (1 row)
```

Vẫn còn một cách khác để sử dụng một hàm mà trả về một dạng tổng hợp là gọi nó như một hàm của bảng, như được mô tả trong Phần 35.4.7.

35.4.3. Hàm SQL với các tên tham số

Là có khả năng để gắn các tên tới các tham số của một hàm, ví dụ

```
CREATE FUNCTION tf1 (acct_no integer, debit numeric) RETURNS numeric AS $$
        UPDATE bank
```

```
                SET balance = balance - $2
                WHERE accountno = $1
        RETURNING balance;
$$ LANGUAGE SQL;
```

Đây là tham số đầu tiên có tên acct_no, và tham số thứ 2 có tên debit. Cho tới nay bản thân hàm SQL có liên quan, các tên đó chỉ là sự trang trí; bạn vẫn phải tham chiếu tới các tham số như $1, $2, ... bên trong thân hàm. (Vài ngôn ngữ thủ tục để cho bạn sử dụng các tên tham số thay vào đó). Tuy nhiên, việc gắn các tên tới các tham số là hữu dụng cho mục đích làm tài liệu. Khi hàm có nhiều tham số thì nó cũng hữu dụng để sử dụng các tên đó khi gọi hàm, như được mô tả trong Phần 4.3.

35.4.4. Hàm SQL với các tham số đầu ra

Một cách lựa chọn khác để mô tả các kết quả của một hàm là xác định nó với các tham số đầu ra, như trong ví dụ này:

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;

SELECT add_em(3,7);

add_em
-----------
       10
 (1 row)
```

Về cơ bản điều này không khác với phiên bản add_em được chỉ ra trong Phần 35.4.1. Giá trị thực của các tham số đầu vào là chúng đưa ra một cách thức thuận tiện xác định các hàm trả về vài cột. Ví dụ,

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;

   SELECT * FROM sum_n_product(11,42);

  sum | product
  ------+------------
     53 | 462
   (1 row)
```

Điều gì thực sự đã xảy ra ở đây là chúng ta đã tạo ra một dạng tổng hợp nặc danh cho kết quả của hàm đó. Ví dụ ở trên có kết quả cuối cùng y hệt như

```
CREATE TYPE sum_prod AS (sum int, product int);
CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

nhưng không phải bận tâm với định nghĩa dạng tổng hợp tách biệt thường là thuận tiện. Lưu ý rằng các tên được gắn với các tham số đầu ra không chỉ là trang trí, mà xác định các tên cột của dạng tổng hợp nặc danh đó. (Nếu bạn bỏ qua tên đối với một tham số đầu ra, thì hệ thống sẽ chọn một tên của riêng nó).

Lưu ý rằng các tham số đầu ra không được đưa vào trong việc gọi danh sách đối số khi triệu gọi một hàm như vậy từ SQL. Điều này là vì PostgreSQL xem xét chỉ các tham số đầu vào để xác định chữ ký việc gọi hàm đó. Điều này cũng có nghĩa là chỉ các tham số đầu vào là quan trọng khi tham chiếu tới hàm vì các mục đích như bỏ nó. Chúng ta có thể bỏ hàm ở trên hoặc bằng

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Các tham số có thể được đánh dấu như là IN (mặc định), OUT, INOUT, hoặc VARIADIC. Một tham số INOUT phục vụ như là cả một tham số đầu ra (một phần của lời gọi danh sách đối số) và một tham số đầu ra (một phần của dạng bản ghi kết quả). Các tham số VARIADIC là các tham số đầu vào, nhưng được đối xử đặc biệt như được mô tả tiếp sau.

35.4.5. Hàm SQL với các số đối số biến thiên

Các hàm SQL có thể được khai báo để chấp nhận các số đối số biến thiên, miễn là tất cả các đối số "tùy chọn" có dạng dữ liệu y hệt. Các đối số tùy chọn sẽ được truyền tới hàm như một mảng. Hàm đó được khai báo bằng việc làm cho tham số cuối cùng như là VARIADIC; tham số này phải được khai báo như là của một dạng mảng. Ví dụ:

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
        SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);

mleast
----------
      -1
(1 row)
```

Một cách hiệu quả, tất cả các đối số thực sự ở hoặc ngoài vị trí của VARIADIC được tập hợp trong một mảng 1 chiều, dường như bạn đã viết

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]); -- doesn't work (không làm việc)
```

Bạn thực sự không thể viết điều đó, dù - hoặc ít nhất, nó sẽ không khớp với định nghĩa hàm này. Một tham số được đánh dấu VARIADIC khớp với một hoặc nhiều lần hơn dạng phần tử của nó, chứ không phải dạng của riêng nó.

Đôi khi là hữu dụng để có khả năng truyền một mảng được xây dựng rồi tới một hàm biến thiên; điều này đặc biệt thuận tiện khi một hàm biến thiên muốn truyền tham số mảng của nó với nhau. Bạn có thể làm điều đó bằng việc chỉ định VARIADIC trong lời gọi:

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

Điều này ngăn chặn sự mở rộng tham số biến thiên của hàm trong dạng phần tử của nó, bằng cách đó cho phép giá trị đối số mảng khớp bình thường. VARIADIC chỉ có thể được gắn tới đối số thực cuối cùng của một lời gọi hàm.

Các tham số phần tử mảng được sinh ra từ một tham số biến thiên được đối xử như không có bất kỳ tên nào của riêng chúng. Điều này có nghĩa là không có khả năng để gọi một hàm biến thiên bằng việc sử dụng các đối số được đặt tên (Phần 4.3), ngoại trừ khi bạn chỉ định VARIADIC. Ví dụ, điều này sẽ làm việc:

```
SELECT mleast(VARIADIC arr := ARRAY[10, -1, 5, 4.4]);
```

nhưng điều này thì không:

```
SELECT mleast(arr := 10);
SELECT mleast(arr := ARRAY[10, -1, 5, 4.4]);
```

35.4.6. Hàm SQL với các giá trị mặc định cho các đối số

Các hàm có thể được khai báo với các giá trị mặc định cho một vài hoặc tất cả các đối số đầu vào. Các giá trị mặc định được chèn vào bất kỳ khi nào hàm đó được gọi với nhiều đối số thực sự thiếu. Vì các đối số chỉ có thể bị bỏ qua từ cuối của danh sách đối số thực, tất cả các tham số sau một tham số có giá trị mặc định cũng phải có các giá trị mặc định. (Dù sử dụng ký hiệu đối số được đặt tên có thể cho phép hạn chế này được nới lỏng, nó vẫn bị ép buộc sao cho ký hiệu đối số vị trí làm việc một cách hợp lý). Ví dụ:

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
SELECT $1 + $2 + $3;
$$;
          SELECT foo(10, 20, 30);

foo
-----
60
(1 row)

SELECT foo(10, 20);
 foo
-------
  33
(1 row)

SELECT foo(10);
 foo
-------
 15
(1 row)

SELECT foo(); -- fails since there is no default for the first argument
ERROR: function foo() does not exist
```

Dấu bằng (=) cũng có thể được sử dụng tại chỗ đối với từ khóa mặc định DEFAULT.

35.4.7. Hàm SQL như các nguồn bảng

Tất cả các hàm SQL có thể được sử dụng trong mệnh đề FROM của một truy vấn, nhưng nó đặc biệt hữu dụng cho các hàm trả về các dạng tổng hợp. Nếu hàm được xác định để trả về một dạng cơ bản,

thì hàm của bảng tạo ra bảng có 1 cột. Nếu hàm được xác định để trả về một dạng tổng hợp, thì hàm của bảng tạo ra 1 cột cho từng thuộc tính của dạng tổng hợp đó. Đây là một ví dụ:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
        SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

fooid | foosubid | fooname | upper
-------+------------+-------------+---------
    1 |          1 |  Joe        | JOE
(1 row)
```

Như ví dụ chỉ ra, chúng ta có thể làm việc với các cột kết quả của hàm hệt như nếu chúng là các cột của một bảng thông thường.

Lưu ý rằng chúng ta chỉ có 1 hàng của hàm đó. Điều này là vì chúng ta đã không sử dụng SETOF. Điều đó được mô tả trong phần tiếp sau.

35.4.8. Hàm SQL trả về các tập hợp

Khi một hàm SQL được khai báo như là trả về SETOF *sometype*, thì truy vấn cuối cùng của hàm đó được thực thi tới khi kết thúc, và từng hàng mà nó đưa ra kết quả được trả về như một phần tử của tập kết quả.

Tính năng này thường được sử dụng khi gọi hàm trong mệnh đề FROM. Trong trường hợp này, từng hàng được hàm đó trả về trở thành một hàng của bảng được truy vấn đó nhìn thấy. Ví dụ, giả sử bảng foo có các nội dung y hệt như ở trên, và chúng ta nói:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
        SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Sau đó có thể có:

```
fooid | foosubid | fooname
-------+------------+-------------
    1 |          1 | Joe
    1 |          2 | Ed
(2 rows)
```

Cũng có khả năng trả về nhiều hàng với các cột được các tham số đầu ra xác định, như thế này:

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
        SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

```
SELECT * FROM sum_n_product_with_tab(10);

sum | product
------+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

Điểm mấu chốt ở đây là bạn phải viết RETURNS SETOF record để chỉ ra rằng hàm đó trả về nhiều hàng thay vì chỉ 1 hàng. Nếu chỉ có một tham số đầu ra, hãy viết dạng tham số đó thay vì record.

Hiện hành, các hàm trả về các tập hợp cũng có thể được gọi trong danh sách chọn của một truy vấn. Đối với từng hàng mà truy vấn đó tự nó tạo ra, thì tập hợp của hàm trả về được triệu gọi, và một hàng đầu ra được sinh ra cho từng phần tử của tập hợp kết quả của hàm đó. Tuy nhiên, hãy lưu ý, rằng khả năng này không được tán thành và có thể bị loại bỏ trong các phiên bản trong tương lai. Sau đây là một ví dụ hàm trả về một tập hợp từ danh sách chọn:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
        SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM nodes;

   name    | parent
-----------+----------
Top         |
Child1      | Top
Child2      | Top
Child3      | Top
SubChild1 | Child1
SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');

listchildren
--------------
Child1
Child2
Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;

name  | listchildren
---------+--------------
Top     | Child1
Top     | Child2
Top     | Child3
Child1 | SubChild1
Child1 | SubChild2
(5 rows)
```

Trong SELECT cuối cùng, lưu ý rằng không hàng đầu ra nào xuất hiện cho Child2, Child3, ... Điều này

xảy ra vì listchildren trả về một tập rỗng cho các đối số đó, nên không hàng kết quả nào được tạo ra.

> **Lưu ý**: Nếu lệnh cuối của hàm là INSERT, UPDATE, hoặc DELETE với RETURNING, thì lệnh đó sẽ luôn được thực thi tới kết thúc, thậm chí nếu hàm đó không được khai báo bằng SETOF hoặc truy vấn gọi không lấy được tất cả các hàng kết quả. Bất kỳ hàng dư thừa nào được mệnh đề RETURNING tạo ra sẽ âm thầm bị bỏ đi, nhưng các sửa đổi của bảng được ra lệnh vẫn xảy ra (và tất cả được hoàn thành trước khi trả về từ hàm đó).

35.4.9. Hàm SQL trả về bảng

Có một cách khác để khai báo một hàm như là trả về một tập hợp, đó là sử dụng cú pháp RETURNS TABLE(*columns*). Điều này là tương đương với việc sử dụng một hoặc nhiều hơn các tham số OUT cộng với việc đánh dấu hàm đó như là trả về SETOF record (hoặc SETOF một dạng tham số đầu ra duy nhất, một cách phù hợp). Ký hiệu này được chỉ định trong các phiên bản gần đây của tiêu chuẩn SQL, và vì thế có thể khả chuyển hơn so với việc sử dụng SETOF.

Ví dụ, một mẫu ví dụ đứng trước tổng - và - sản phẩm cũng có thể được thực hiện cách này:

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
        SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Không được phép sử dụng các tham số rõ ràng OUT hoặc INOUT với ký hiệu RETURNS TABLE - bạn phải đặt tất cả các cột đầu ra trong danh sách TABLE.

35.4.10. Hàm SQL đa hình

Các hàm SQL có thể được khai báo để chấp nhận và trả về các dạng đa hình anyelement, anyarray, anynonarray, và anyenum. Xem Phần 35.2.5 để có giải thích chi tiết hơn về các hàm đa hình. Đây là một hàm đa hình make_array mà xây dựng một mảng từ 2 phần tử dạng dữ liệu tùy ý:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
        SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;

intarray | textarray
-----------+-----------
  {1,2} | {a,b}
(1 row)
```

Hãy lưu ý sử dụng đoạn gõ phím 'a'::text để chỉ định rằng đối số đó là dạng text. Điều này được yêu cầu nếu đối số đó chỉ là một hằng chuỗi, vì nếu không thì nó có thể được đối xử như là dạng unknown, và mảng của unknown thì không phải là một dạng hợp lệ. Không có đoạn gõ phím đó, thì bạn sẽ gặp các lỗi như thế này:

```
ERROR: could not determine polymorphic type because input has type "unknown"
```

Là được phép để có các đối số đa hình với một dạng trả về cố định, nhưng điều nghịch đảo thì không. Ví dụ:

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
        SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);

is_greater
--------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
        SELECT 1;
$$ LANGUAGE SQL;
ERROR: cannot determine result data type
DETAIL: A function returning a polymorphic type must have at least one polymorphic argument.
```

Đa hình có thể được sử dụng với các hàm có các đối số đầu ra. Ví dụ:

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);

 f2 | f3
----+---------
22 | {22,22}
(1 row)
```

Đa hình cũng có thể được sử dụng với các hàm biến thiên. Ví dụ:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
        SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);

anyleast
-------------
         -1
(1 row)

SELECT anyleast('abc'::text, 'def');

anyleast
------------
abc
(1 row)


CREATE FUNCTION concat(text, VARIADIC anyarray) RETURNS text AS $$
        SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat('|', 1, 4, 2);

 concat
-----------
 1 | 4| 2
(1 row)
```

35.5. Hàm quá tải

Hơn một hàm có thể được xác định với tên SQL y hệt, miễn là các đối số chúng lấy là khác nhau. Nói cách khác, các tên hàm có thể *chất quá tải (overload)*. Khi truy vấn được thực thi, máy chủ sẽ xác định hàm nào để gọi từ các dạng dữ liệu và số lượng các đối số được cung cấp. Việc chất quá tải cũng có thể được sử dụng để mô phỏng các hàm với vài đối số biến đổi, tới một số tối đa hữu hạn.

Khi tạo họ các hàm quá tải, nên cẩn thận không tạo ra sự mù mờ tối nghĩa. Ví dụ, đưa ra các hàm:

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

không rõ ngay lập tức hàm nào có thể được gọi với vài đầu vào tầm thường như `test(1, 1.5)`. Các qui tắc của giải pháp được triển khai hiện hành được mô tả trong Chương 10, nhưng là không khôn ngoan để thiết kế một hệ thống mà tế nhị dựa vào hành vi này.

Hàm mà lấy một đối số duy nhất của dạng tổng hợp thường sẽ không có tên y hệt như bất kỳ thuộc tính (trường) nào của dạng đó. Hãy nhớ rằng `attribute(table)` được cân nhắc là tương đương với `table.attribute`. Trong trường hợp có một sự mù mờ giữa một hàm ở dạng tổng hợp và một thuộc tính dạng tổng hợp, thì thuộc tính đó sẽ luôn được sử dụng. Là có khả năng để ghi đè sự lựa chọn đó bằng việc lượng hóa sơ đồ tên hàm (đó là, `schema.func(table)`) nhưng là tốt hơn để tránh vấn đề này bằng việc không chọn các tên xung đột.

Xung đột có khả năng khác là giữa các hàm biến thiên và không biến thiên. Ví dụ, là có khả năng để tạo cả `foo(numeric)` và `foo(VARIADIC numeric[])`. Trong trường hợp này, là chưa rõ hàm nào sẽ phù hợp với lời gọi cung cấp một đối số có một chữ số duy nhất, như `foo(10.1)`. Qui tắc là hàm đó xuất hiện sớm hơn trong đường tìm kiếm được sử dụng, hoặc nếu 2 hàm là cùng trong sơ đồ y hệt, thì hàm không biến thiên sẽ được ưu tiên.

Khi làm quá tải các hàm ngôn ngữ C, có một ràng buộc bổ sung thêm: Tên của C đối với từng hàm trong họ các hàm bị quá tải phải khác với các tên của C đối với tất cả các hàm khác, hoặc được tải nội bộ hoặc động. Nếu qui tắc này bị vi phạm, thì hành vi đó là không khả chuyển. Bạn có thể có một lỗi liên kết thời gian chạy, hoặc một trong các hàm sẽ được gọi (thường là hàm nội bộ). Dạng lựa chọn của mệnh đề `AS` cho lệnh SQL `CREATE FUNCTION` tách riêng tên hàm SQL ra khỏi tên hàm trong mã nguồn C. Ví dụ:

```
CREATE FUNCTION test(int) RETURNS int
        AS 'filename', 'test_1arg'
        LANGUAGE C;

CREATE FUNCTION test(int, int) RETURNS int
        AS 'filename', 'test_2arg'
        LANGUAGE C;
```

Tên của các hàm C ở đây phản ánh một trong nhiều qui ước có khả năng.

35.6. Chủng loại dễ biến động của hàm

Mỗi hàm có một sự phân loại *dễ biến động (volatility)*, với các khả năng là `VOLATILE`, `STABLE`, hoặc

IMMUTABLE. VOLATILE là mặc định nếu lệnh CREATE FUNCTION không chỉ định một chủng loại. Chủng loại dễ biến động là một hứa hẹn cho trình tối ưu hóa về hành vi của hàm:

- Hàm VOLATILE có thể làm mọi điều, bao gồm việc sửa đổi cơ sở dữ liệu. Nó có thể trả về các kết quả khác nhau trong các lời gọi liên tục với các đối số y hệt. Trình tối ưu hóa không giả thiết về hành vi của các hàm như vậy. Truy vấn sử dụng một hàm dễ biến động sẽ đánh giá lại hàm đó ở từng hàng, nơi mà giá trị của nó là cần thiết.
- Hàm STABLE không thể sửa đổi cơ sở dữ liệu và được đảm bảo trả về các kết quả y hệt, đưa ra các đối số y hệt cho tất cả các hàm bên trong một lệnh duy nhất. Chủng loại này cho phép trình tối ưu hóa tối ưu nhiều lời gọi hàm trong một lời gọi duy nhất. Đặc biệt, là an toàn để sử dụng một biểu thức có chứa một hàm như vậy trong điều kiện quét chỉ số. (Vì một sự quét chỉ số sẽ đánh giá giá trị so sánh chỉ một lần, chứ không phải mỗi lần ở từng hàng, nên là không hợp lệ để sử dụng hàm VOLATILE trong một điều kiện quét chỉ số).
- Hàm IMMUTABLE không thể sửa đổi cơ sở dữ liệu và được đảm bảo để trả về các kết quả y hệt, đưa ra các đối số y hệt vĩnh viễn. Chủng loại này cho phép trình tối ưu hóa đánh giá trước hàm đó khi một truy vấn gọi nó bằng các đối số hằng. Ví dụ, một truy vấn như SELECT ... WHERE x = 2 + 2 có thể được đơn giản hóa thành SELECT ... WHERE x = 4, vì hàm nằm bên dưới toán tử cộng số nguyên được đánh dấu là IMMUTABLE.

Vì các kết quả tối ưu hóa tốt nhất, bạn nên gắn nhãn cho các hàm của bạn bằng chủng loại dễ biến động khắt khe nhất mà là hợp lệ đối với chúng.

Bất kỳ hàm nào với các hiệu ứng phụ cũng phải được gắn nhãn VOLATILE, sao cho các lời gọi tới nó không thể được tối ưu hóa. Thậm chí một hàm không với các hiệu ứng phụ cần phải được gắn nhãn VOLATILE nếu giá trị của nó có thể thay đổi bên trong một truy vấn duy nhất; vài ví dụ là random(), currval(), timeofday().

Một ví dụ quan trọng khác là họ các hàm current_timestamp định lượng như là STABLE, vì các giá trị của chúng không thay đổi trong một giao dịch.

Có sự khác biệt khá nhỏ giữa các chủng loại STABLE và IMMUTABLE khi xem xét các truy vấn tương tác đơn giản được lên kế hoạch và ngay lập tức được thực thi: không thành vấn đề nhiều bất kể một hàm được thực thi hay không một khi trong quá trình lên kế hoạch hoặc một khi trong quá trình khởi động sự thực thi truy vấn. Nhưng có sự khác biệt lớn nếu kế hoạch đó được lưu giữ và sử dụng lại sau đó. Việc gắn nhãn cho một hàm IMMUTABLE khi nó thực sự không phải có thể cho phép nó được khoanh lại sớm hơn về một hằng số trong quá trình lên kế hoạch, gây ra một giá trị cũ đang được sử dụng lại trong quá trình các sử dụng tiếp theo của kế hoạch. Đây là sự may rủi khi sử dụng các lệnh được chuẩn bị sẵn hoặc khi sử dụng các ngôn ngữ hàm trữ các kế hoạch (như PL/pgSQL).

Đối với các hàm được viết trong SQL hoặc trong bất kỳ ngôn ngữ thủ tục tiêu chuẩn nào, có một đặc tính quan trọng thứ 2 được chủng loại dễ biến động xác định, ấy là tính trực quan của bất kỳ sự thay đổi dữ liệu nào mà từng được lệnh SQL thực hiện mà đang gọi hàm đó. Hàm VOLATILE sẽ thấy những thay đổi như vậy, hàm STABLE hoặc IMMUTABLE thì sẽ không. Hành vi này được triển khai bằng việc sử dụng hành vi chụp nhanh hình MVCC (xem Chương 13); các hàm STABLE và

IMMUTABLE sử dụng một hình chụp nhanh được thiết lập ngay từ đầu của việc gọi truy vấn, trong đó các hàm VOLATILE giành được hình chụp nhanh tươi mới ở đầu của từng truy vấn mà nó thực thi.

> **Lưu ý**: Các hàm được viết trong C có thể quản lý các hình chụp nhanh theo cách mà chúng muốn, nhưng thường là ý tưởng tốt để cũng làm cho các hàm C làm việc theo cách đó.

Vì hành vi chụp nhanh này, hàm chỉ chứa lệnh SELECT có thể an toàn được đánh dấu là STABLE, thậm chí nếu nó lựa chọn từ các bảng có thể đang diễn ra các sửa đổi từ các truy vấn hiện hành. PostgreSQL sẽ thực thi tất cả các lệnh của một hàm STABLE bằng việc sử dụng hình chụp nhanh được thiết lập cho việc gọi truy vấn, và vì thế nó sẽ thấy một kiểu nhìn cố định của cơ sở dữ liệu qua truy vấn đó.

Hành vi chụp hình nhanh y hệt được sử dụng cho các lệnh SELECT trong các hàm IMMUTABLE. Thường là không khôn ngoan để lựa chọn từ các bảng cơ sở dữ liệu bên trong hàm IMMUTABLE, vì tính không thay đổi sẽ bị đổ vỡ nếu các nội dung của bảng thay đổi. Tuy nhiên, PostgreSQL không ép điều bạn không làm đó.

Lỗi phổ biến là gắn nhãn cho một hàm IMMUTABLE khi các kết quả của nó phụ thuộc vào một tham số cấu hình. Ví dụ, một hàm điều khiển các dấu thời gian có thể có các kết quả dựa vào thiết lập vùng thời gian. Vì sự an toàn, các hàm như vậy sẽ được gắn nhãn STABLE.

> **Lưu ý**: Trước PostgreSQL phiên bản 8.0, yêu cầu rằng các hàm STABLE và IMMUTABLE không thể sửa đổi cơ sở dữ liệu đã không bị hệ thống ép buộc. Các phiên bản 8.0 và sau này ép nó bằng việc yêu cầu các hàm SQL và các hàm ngôn ngữ thủ tục của các chủng loại đó không chứa các lệnh SQL khác với lệnh SELECT. (Đây không phải là một kiểm thử báo có dự kiến hoàn chỉnh, vì các hàm như vậy có thể vẫn gọi các hàm VOLATILE mà sửa đổi cơ sở dữ liệu. Nếu bạn làm thế, bạn sẽ thấy rằng hàm STABLE hoặc IMMUTABLE không lưu ý các thay đổi của cơ sở dữ liệu được hàm được gọi đó áp dụng, vì chúng bị ẩn khỏi hình chụp nhanh của nó).

35.7. Hàm ngôn ngữ thủ tục

PostgreSQL cho phép các hàm do người sử dụng định nghĩa sẽ được viết trong các ngôn ngữ khác ngoài SQL và C. Các ngôn ngữ khác đó thường được gọi là các *ngôn ngữ thủ tục* - PL (Procedural Language). Các ngôn ngữ thủ tục sẽ không được xây dựng trong máy chủ PostgreSQL; chúng được các module có khả năng tải được chào. Xem Chương 38 và các chương sau để có thêm thông tin.

35.8. Hàm nội bộ

Các hàm nội bộ là các hàm được viết trong C mà về mặt thống kê từng được liên kết trong máy chủ PostgreSQL. “Thân” của định nghĩa hàm đó chỉ định tên ngôn ngữ C của hàm, điều cần phải không là y hệt như tên đang được khai báo để sử dụng SQL. (Vì các lý do của sự tương thích ngược, một thân rỗng được chấp nhận như thể tên hàm ngôn ngữ C là tên như tên của SQL).

Thông thường, tất cả các hàm nội bộ thể hiện trong máy chủ được khai báo khi khởi tạo bó cơ sở dữ liệu (xem Phần 17.2), nhưng một người sử dụng có thể sử dụng CREATE FUNCTION để tạo các tên

hiệu bổ sung thêm cho một hàm nội bộ. Các hàm nội bộ được khai báo trong CREATE FUNCTION với tên ngôn ngữ internal. Ví dụ, để tạo một tên hiệu cho hàm sqrt:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Hầu hết các hàm nội bộ kỳ vọng được khai báo là khắt khe "strict").

> **Lưu ý**: Không phải tất cả các hàm "được định nghĩa trước" là các hàm "nội bộ" theo nghĩa nêu trên. Vài hàm được định nghĩa trước được viết trong SQL.

35.9. Hàm ngôn ngữ C

Các hàm do người sử dụng định nghĩa có thể được viết trong C (hoặc một ngôn ngữ mà có thể được làm cho tương thích với C, như C++). Các hàm như vậy được biên dịch trong các đối tượng có khả năng tải được động (còn được gọi là các thư viện được chia sẻ) và được máy chủ tải lên theo yêu cầu. Tính năng tải động là những gì làm cho các hàm "ngôn ngữ C" khác biệt với các hàm "nội bộ" - các qui ước lập trình thực tế về cơ bản là y như nhau cho cả 2 dạng. (Vì thế, thư viện hàm nội bộ tiêu chuẩn là một nguồn giàu có các ví dụ lập trình cho các hàm C do người sử dụng định nghĩa).

2 qui ước gọi khác nhau hiện được sử dụng cho các hàm C. Qui ước gọi "phiên bản 1" mới hơn được chỉ định bằng cách viết một lời gọi macro PG_FUNCTION_INFO_V1() cho hàm đó, như được minh họa ở bên dưới. Thiếu một macro như vậy chỉ ra hàm ("phiên bản 0") kiểu cũ. Tên ngôn ngữ được chỉ định trong CREATE FUNCTION là C trong cả 2 trường hợp. Các hàm kiểu cũ bây giờ không được tán thành vì các vấn đề về tính khả chuyển và thiếu chức năng, nhưng chúng vẫn còn được hỗ trợ vì các lý do về tính tương thích.

35.9.1. Tải động

Lần đầu một hàm do người sử dụng định nghĩa trong một tệp đối tượng có khả năng tải được đặc biệt được gọi là một phiên, trình tải động sẽ tải tệp đối tượng đó vào bộ nhớ sao cho hàm đó có thể được gọi. CREATE FUNCTION cho một hàm C do người sử dụng định nghĩa vì thế phải chỉ định 2 mẩu thông tin cho hàm đó: tên và tệp đối tượng có khả năng tải được, và tên C (ký hiệu liên kết) của hàm đặc thù để gọi bên trong tệp đối tượng đó. Nếu tên C không được chỉ định rõ ràng thì nó được giả thiết sẽ là y hệt như tên hàm SQL.

Thuật toán sau đây được sử dụng để định vị tệp đối tượng được chia sẻ dựa vào tên được đưa ra trong lệnh CREATE FUNCTION:

1. Nếu tên là một đường dẫn tuyệt đối, thì tệp được đưa ra được tải.
2. Nếu tên bắt đầu với chuỗi $libdir, thì phần đó được thay thế bằng tên thư mục thư viện gói PostgreSQL, được xác định khi xây dựng.
3. Nếu tên đó không chứa phần thư mục, thì tệp đó được tìm kiếm trong đường dẫn được biến cấu hình dynamic_library_path chỉ định.

4. Nếu không thì (tệp đó đã không được tìm thấy để lấy tên như được đưa ra, điều có khả năng sẽ hỏng. (Là không đáng tin để dựa vào thư mục làm việc hiện hành).

Nếu tuần tự này không làm việc, thì phần mở rộng tên tệp thư viện được chia sẻ đặc thù nền tảng (thường là .so) được nối thêm vào tên được đưa ra và tuần tự này sẽ được cố làm một lần nữa. Nếu nó cũng lại thất bại, thì tải sẽ hỏng.

Được khuyến cáo để định vị các thư viện được chia sẻ hoặc tương đối với $libdir hoặc thông qua đường dẫn thư viện động. Điều này đơn giản hóa các bản cập nhật phiên bản nếu cài đặt mới là ở một vị trí khác. Thư mục thực $libdir nằm trong đó có thể được thấy bằng lệnh pg_config -pkglibdir.

Mã người sử dụng (user ID) mà máy chủ PostgreSQL quản lý phải có khả năng đi qua đường dẫn tới tệp bạn định tải. Làm cho tệp hoặc thư mục mức cao hơn không có khả năng đọc được và/hoặc không thực thi được đối với người sử dụng postgres là một sai lầm phổ biến.

Trong bất kỳ trường hợp nào, tên tệp được đưa ra trong lệnh CREATE FUNCTION được ghi lại thực trong các catalog hệ thống, như thể tệp đó cần được tải lên một lần nữa thủ tục y hệt được áp dụng.

> **Lưu ý**: PostgreSQL sẽ không biên dịch một hàm C một cách tự động. Tệp đối tượng đó phải được biên dịch trước khi nó được tham chiếu trong một lệnh CREATE FUNCTION. Xem Phần 35.9.6 để có thêm thông tin.

Để đảm bảo rằng một tệp đối tượng được tải động không bị tải vào trong một máy chủ không tương thích, PostgreSQL kiểm tra xem tệp đó có chứa một "khối ma thuật" (magic block) với các nội dung phù hợp hay không. Điều này cho phép máy chủ dò tìm ra các chỗ không tương thích rõ ràng, như mã được biên dịch cho một phiên bản chính khác của PostgreSQL. Khối ma thuật đó được yêu cầu như đối với PostgreSQL 8.2. Để đưa vào một khối ma thuật, hãy viết điều này vào một (và chỉ một) trong số các tệp nguồn của module đó, sau khi đã đưa vào đầu đề fmgr.h:

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

Kiểm thử #ifdef có thể bị bỏ qua nếu mã không cần biên dịch đối với các phiên bản PostgreSQL trước 8.2.

Sau khi nó được sử dụng lần đầu, một tệp đối tượng được tải lên tự động vẫn nằm trong bộ nhớ. Các lời gọi trong tương lai trong cùng một phiên làm việc tới (các) hàm trong tệp đó sẽ chỉ chịu tổng chi phí nhỏ của một sự tra cứu bảng các ký hiệu. Nếu bạn cần phải ép buộc một sự tải lại một tệp đối tượng, ví dụ sau khi tái biên dịch nó, thì hãy bắt đầu một phiên làm việc mới.

Tùy ý lựa chọn, một tệp được tải động có thể chứa các hàm khởi xướng và kết thúc. Nếu tệp đó bao gồm một hàm có tên là _PG_init, thì hàm đó sẽ được gọi ngay lập tức sau khi tải tệp đó. Hàm đó không nhận tham số nào và sẽ trả về void. Nếu tệp đó bao gồm một hàm có tên là _PG_fini, thì hàm đó sẽ được gọi ngay lập tức trước khi dỡ bỏ tải tệp đó. Cũng vậy, hàm đó không nhận các tham số và sẽ trả về rỗng (void). Lưu ý rằng _PG_fini sẽ chỉ được gọi trong quá trình một sự bỏ tải tệp, chứ không trong quá trình kết thúc. (Hiện tại, sự bỏ tải bị vô hiệu hóa và sẽ không bao giờ xảy ra, nhưng điều này có thể thay đổi trong tương lai).

35.9.2. Dạng cơ bản trong các hàm ngôn ngữ C

Để biết cách viết các hàm ngôn ngữ C, bạn cần phải biết cách PostgreSQL trình bày nội bộ các dạng dữ liệu cơ bản và cách mà chúng có thể được truyền tới và từ các hàm. Trong nội bộ, PostgreSQL coi một dạng cơ bản như là một "khối bộ nhớ". Các hàm do người sử dụng định nghĩa mà bạn xác định qua dạng thay vì xác định cách mà PostgreSQL có thể vận hành trong nó. Đó là, PostgreSQL sẽ chỉ lưu trữ và truy xuất dữ liệu từ đĩa và sử dụng các hàm do người sử dụng của bạn định nghĩa cho các dữ liệu đầu vào, qui trình và đầu ra.

Các dạng cơ bản có thể có 1 trong 3 định dạng nội bộ sau:

- truyền bằng giá trị, độ dài cố định
- truyền bằng tham chiếu, độ dài cố định
- truyền bằng tham chiếu, độ dài biến đổi

Các dạng theo giá trị chỉ có thể là 1, 2 hoặc 4 byte độ dài (cũng có cả 8 byte, nếu sizeof(Datum) là 8 trên máy của bạn). Bạn nên cẩn thận xác định các dạng của bạn sao cho chúng sẽ có kích cỡ y hệt nhau (theo byte) trên tất cả các cấu trúc. Ví dụ, dạng long là nguy hiểm vì nó là 4 byte trên một số máy và 8 byte trên một số máy khác, trong khi dạng int là 4 byte trên hầu hết các máy Unix. Triển khai hợp lý dạng int4 trên các máy Unix có lẽ là:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

Mặc khác, các dạng độ dài cố định của bất kỳ kích cỡ nào cũng có thể được truyền bằng tham chiếu. Ví dụ, đây là một triển khai mẫu của một dạng PostgreSQL:

```
/* 16-byte structure, passed by reference */
typedef struct
{
        double x, y;
} Point;
```

Chỉ các con trỏ tới các dạng như vậy có thể được sử dụng khi truyền chúng vào và ra đối với các hàm PostgreSQL. Để trả về một giá trị một dạng như vậy, hãy phân bổ số lượng quyền của bộ nhớ với palloc, điền vào bộ nhớ được phân bổ, và trả về một con trỏ cho nó. (Hơn nữa, nếu bạn chỉ muốn trả về giá trị y hệt như một giá trị của các đối số đầu vào của bạn của dạng dữ liệu y hệt, thì bạn có thể bỏ qua palloc dư thừa và chỉ trả về con trỏ cho giá trị đầu vào đó).

Cuối cùng, tất cả các dạng có độ dài biến đổi cũng phải được tham chiếu truyền. Tất cả các dạng có độ dài biến đổi phải bắt đầu bằng một trường độ dài chính xác 4 byte, và tất cả các dữ liệu phải được lưu trữ bên trong dạng đó phải nằm trong bộ nhớ ngay lập tức sau trường độ dài đó. Trường độ dài đó có chứa tổng độ dài của cấu trúc đó, đó là, nó bao gồm kích thước của bản thân trường độ đài đó.

Điểm quan trọng khác phải tránh để bất kỳ bit không được khởi tạo nào bên trong các giá trị dạng dữ liệu; ví dụ, hãy chăm sóc để zero (0) ra ngoài bất kỳ byte chêm vào phù hợp nào mà có thể được trình bày trong các cấu trúc. Không có điều này, các hằng tương đương về mặt logic của dạng dữ

liệu của bạn có thể được xem là không bằng đối với trình lên kế hoạch, dẫn tới các kế hoạch không có hiệu quả (dù không phải là không đúng).

> **Cảnh báo**
>
> Không bao giờ sửa đổi nội dung của một giá trị đầu vào được truyền bằng tham chiếu. Nếu bạn làm thế thì bạn có khả năng làm hỏng dữ liệu trên đĩa, vì con trỏ mà bạn đã đưa ra có thể trỏ trực tiếp vào một bộ nhớ tạm của đĩa. Ngoại lệ duy nhất cho qui tắc này được giải thích trong Phần 35.10.

Như một ví dụ, chúng ta có thể xác định văn bản dạng như sau:

```
typedef struct {
        int4 length;
        char data[1];
} text;
```

Rõ ràng, trường dữ liệu được khai báo ở đây là không đủ dài để giữ tất cả các chuỗi có khả năng. Vì không có khả năng để khai báo một cấu trúc kích cỡ biến thiên trong C, chúng ta dựa vào tri thức rằng trình biên dịch C sẽ không kiểm tra dải các chỉ số dưới của mảng. Chúng ta chỉ phân bổ lượng không gian cần thiết và sau đó truy cập mảng đó như thể nó đã được khai báo độ dài đúng. (Đây là một mẹo phổ biến mà bạn có thể đọc trong nhiều sách về C).

Khi điều khiển các dạng độ dài biến động, chúng ta phải cẩn thận phân bổ lượng bộ nhớ đúng và thiết lập trường độ dài đúng. Ví dụ, nếu chúng ta muốn lưu trữ 40 byte trong một cấu trúc văn bản, thì chúng ta có thể sử dụng một phân đoạn mã giống thế này:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ là y hệt như sizeof(int4), nhưng đây là kiểu tốt được xem xét để sử dụng macro VARHDRSZ để tham chiếu tới kích cỡ của tổng chi phí cho một dạng độ dài biến thiên. Hơn nữa, trường độ dài phải được thiết lập bằng việc sử dụng macro SET_VARSIZE, chứ không bằng sự chỉ định đơn giản.

Bảng 35-1 chỉ định dạng C nào tương xứng với dạng SQL nào khi viết một hàm ngôn ngữ C mà sử dụng một dạng của PostgreSQL được xây dựng sẵn. Cột "Được định nghĩa trong" (Defined In) đưa ra tệp đầu đề cần phải được đưa vào để có được định nghĩa dạng đó. (Định nghĩa thực sự có thể là trong một tệp khác mà được tệp được liệt kê đó đưa vào. Được khuyến cáo rằng người sử dụng gắn vào giao diện được định nghĩa đó). Lưu ý rằng bạn luôn nên đưa vào postgres.h trước tiên trong bất kỳ tệp nguồn nào, vì nó khai báo một số điều mà bạn sẽ cần dù là cách nào.

Bảng 35-1. Các dạng C tương đương cho các dạng SQL được xây dựng sẵn

Dạng SQL	Dạng C	Được định nghĩa trong
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h (maybe compiler built-in)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 or int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 or int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	utils/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	utils/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Bây giờ chúng ta đã đi qua tất cả các cấu trúc có khả năng đối với các dạng cơ bản, chúng ta có thể chỉ ra vài ví dụ về các hàm thực.

35.9.3. Các qui ước gọi phiên bản 0

Chúng ta trình bày trước tiên qui ước gọi “kiểu cũ” - dù tiếp cận này bây giờ đã lỗi thời, là dễ dàng hơn để có được sự điều khiển lúc ban đầu. Theo phương pháp phiên bản 0, các đối số và kết quả của

hàm C chỉ được khai báo ở dạng C thông thường, nhưng hãy cẩn thận sử dụng đại diện C của từng dạng dữ liệu SQL như được chỉ ra ở trên.

Đây là một vài ví dụ:

```
#include "postgres.h"
#include <string.h>
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */

int
add_one(int arg)
{
        return arg + 1;
}

/* by reference, fixed length */

float8 *
add_one_float8(float8 *arg)
{
        float8 *result = (float8 *) palloc(sizeof(float8));
        *result = *arg + 1.0;
        return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
        Point *new_point = (Point *) palloc(sizeof(Point));

        new_point->x = pointx->x;
        new_point->y = pointy->y;

        return new_point;
}

/* by reference, variable length */

text *
copytext(text *t)
{
        /*
        * VARSIZE is the total size of the struct in bytes.
        */
        text *new_t = (text *) palloc(VARSIZE(t));
        SET_VARSIZE(new_t, VARSIZE(t));
        /*
        * VARDATA is a pointer to the data region of the struct.
        */
        memcpy((void *) VARDATA(new_t), /* destination */
                (void *) VARDATA(t), /* source */
                VARSIZE(t) - VARHDRSZ); /* how many bytes */
        return new_t;
}

text *
```

```
        concat_text(text *arg1, text *arg2)
        {
        int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
        text *new_text = (text *) palloc(new_text_size);
        SET_VARSIZE(new_text, new_text_size);
        memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
        memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
                VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
        return new_text;
}
```

Giả sử mã ở trên đã được chuẩn bị trong tệp funcs.c và được biên dịch trong một đối tượng chia sẻ, chúng ta có thể định nghĩa các hàm cho PostgreSQL với các lệnh giống thế này:

```
CREATE FUNCTION add_one(integer) RETURNS integer
        AS 'DIRECTORY/funcs', 'add_one'
        LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
        AS 'DIRECTORY/funcs', 'add_one_float8'
        LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
        AS 'DIRECTORY/funcs', 'makepoint'
        LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
        AS 'DIRECTORY/funcs', 'copytext'
        LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
        AS 'DIRECTORY/funcs', 'concat_text'
        LANGUAGE C STRICT;
```

Ở đây, *DIRECTORY* nghĩa là thư mục của tệp thư viện được chia sẻ (ví dụ thư mục chỉ dẫn của PostgreSQL, có chứa mã cho các ví dụ được sử dụng trong phần này). (Kiểu tốt hơn có thể được sử dụng chỉ 'funcs' trong mệnh đề AS, sau khi đã thêm *DIRECTORY* vào đường tìm kiếm. Trong bất kỳ trường hợp nào, chúng ta cũng có thể bỏ qua mở rộng đặc thù hệ thống cho một thư viện được chia sẻ, thường là .so hoặc .sl).

Lưu ý rằng chúng ta đã chỉ định các hàm như là "khắt khe" (strict), nghĩa là hệ thống sẽ tự động giả thiết một kết quả null nếu bất kỳ giá trị đầu vào nào là null. Bằng việc làm này, chúng ta tránh được việc phải kiểm tra các đầu vào null trong mã của hàm. Không có điều này, chúng ta có thể phải kiểm tra các giá trị null một cách rõ ràng, bằng việc kiểm tra một con trỏ null cho từng đối số truyền tham chiếu. (Đối với các đối số truyền giá trị, chúng ta thậm chí không có cách nào để kiểm tra!).

Dù qui ước gọi này là đơn giản để sử dụng, nó là rất không khả chuyển; trong một số kiến trúc thì cách đơn giản để trả về một kết quả null, không phải vượt qua các đối số null theo bất kỳ cách gì khác hơn việc tạo hàm khắt khe. Qui ước phiên bản 1, được trình bày sau đây, vượt qua được các khó khăn đó.

35.9.4. Qui ước gọi phiên bản 1

Qui ước gọi phiên bản 1 dựa vào macro để triệt tiêu hầu hết sự phức tạp của việc truyền các đối số và các kết quả. Khai báo C của một hàm phiên bản 1 luôn là:

```
Datum funcname(PG_FUNCTION_ARGS)
```

Hơn nữa, lời gọi macro:

```
PG_FUNCTION_INFO_V1(funcname);
```

phải xuất hiện trong tệp nguồn y hệt. (Theo qui ước, nó được viết ngay trước bản thân hàm đó). Lời gọi macro này là không cần thiết cho các hàm ngôn ngữ nội bộ, vì PostgreSQL giả thiết rằng tất cả các hàm nội bộ sử dụng qui ước phiên bản 1. Tuy nhiên, nó được yêu cầu cho các hàm tải động.

Trong một hàm phiên bản 1, từng đối số thực được lấy về bằng việc sử dụng một macro PG_GETARG_xxx() mà tương ứng với dạng dữ liệu của đối số đó, và kết quả được trả về bằng việc sử dụng một macro PG_RETURN_xxx() cho dạng trả về đó. PG_GETARG_xxx() lấy như đối số của nó là số đối số của hàm để lấy về, nơi mà sự đếm bắt đầu từ 0. PG_RETURN_xxx() lấy như đối số của nó là giá trị thực để trả về.

Ở đây chúng ta chỉ ra các hàm y hệt như ở bên trên, được mã ở dạng phiên bản 1:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
        int32 arg = PG_GETARG_INT32(0);

        PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
        /* The macros for FLOAT8 hide its pass-by-reference nature. */
        float8 arg = PG_GETARG_FLOAT8(0);

        PG_RETURN_FLOAT8(arg + 1.0);
}


PG_FUNCTION_INFO_V1(makepoint);
```

```
Datum
makepoint(PG_FUNCTION_ARGS)
{
        /* Here, the pass-by-reference nature of Point is not hidden. */
        Point *pointx = PG_GETARG_POINT_P(0);
        Point *pointy = PG_GETARG_POINT_P(1);
        Point *new_point = (Point *) palloc(sizeof(Point));

        new_point->x = pointx->x;
        new_point->y = pointy->y;

        PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
        text *t = PG_GETARG_TEXT_P(0);
        /*
        * VARSIZE is the total size of the struct in bytes.
        */
        text *new_t = (text *) palloc(VARSIZE(t));
        SET_VARSIZE(new_t, VARSIZE(t));
        /*
        * VARDATA is a pointer to the data region of the struct.
        */
        memcpy((void *) VARDATA(new_t), /* destination */
                (void *) VARDATA(t), /* source */
                VARSIZE(t) - VARHDRSZ); /* how many bytes */
        PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
        text *arg1 = PG_GETARG_TEXT_P(0);
        text *arg2 = PG_GETARG_TEXT_P(1);
        int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
        text *new_text = (text *) palloc(new_text_size);

        SET_VARSIZE(new_text, new_text_size);
        memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
        memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
                VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
        PG_RETURN_TEXT_P(new_text);
}
```

Các lệnh CREATE FUNCTION là y hệt như với các lệnh tương đương phiên bản 0.

Nhìn thoáng qua, các qui ước mã phiên bản 1 có lẽ dường như là chính sách ngu ngốc không có điểm gì cả. Tuy nhiên, chúng đưa ra một số cải tiến, vì các macro có thể ẩn dấu chi tiết không cần thiết. Một ví dụ là trong việc mã add_one_float8, chúng ta không còn cần phải nhận thức rằng float8 là một dạng truyền tham chiếu. Một ví dụ khác là các macro GETARG cho các dạng độ dài biến thiên cho phép việc lấy về có hiệu quả hơn các giá trị "được nướng" (được nén hoặc nằm ngoài dòng).

Một cải tiến lớn trong các hàm phiên bản 1 là việc điều khiển tốt hơn các đầu vào và kết quả null. Macro PG_ARGISNULL(n) cho phép một hàm kiểm thử liệu từng đầu vào có là null hay không. (Tất nhiên, làm điều này chỉ khi cần thiết trong các hàm không được khai báo là "khắt khe" (strict). Như với các macro PG_GETARG_xxx(), các đối số đầu vào được tính bắt đầu từ zero. Lưu ý rằng người ta có thể kìm lại việc thực thi PG_GETARG_xxx() cho tới khi người ta đã kiểm tra hợp lệ rằng đối số đó không null. Để trả về một kết quả null, hãy thực thi PG_RETURN_NULL(); điều này làm việc trong cả các hàm khắt khe và không khắt khe.

Các lựa chọn khác được đưa ra trong giao diện dạng mới là 2 phương án của các macro PG_GETARG_xxx(). Phương án đầu, PG_GETARG_xxx_COPY(), đảm bảo trả về một bản sao đối số được chỉ định mà là an toàn cho việc viết vào. (Các macro thông thường đôi lúc sẽ trả về một con trỏ về một giá trị mà được lưu trữ vật lý trong bảng, nó phải không được viết vào. Bằng việc sử dụng các macro PG_GETARG_xxx_SLICE() mà lấy 3 đối số. Đối số đầu là số đối số của hàm (như ở trên). Đối số thứ 2 và 3 là sự bù trừ và độ dài của phân đoạn sẽ được trả về. Các bù trừ được đếm từ zero, và một độ dài âm yêu cầu rằng phần còn lại của giá trị đó sẽ được trả về. Các macro đó đưa ra sự truy cập có hiệu quả hơn tới các phần của các giá trị lớn trong trường hợp nơi mà chúng có dạng lưu trữ "ngoài" (external). (Dạng lưu trữ của cột có thể được chỉ định bằng việc sử dụng ALTER TABLE tablename ALTER COLUMN colname SET STORAGE storagetype. storagetype là một trong số plain, external, extended, hoặc main).

Cuối cùng, các qui ước gọi hàm phiên bản 1 làm cho nó có khả năng trả về tập hợp các kết quả (Phần 35.9.10) và triển khai các hàm trigger (Chương 36) và các điều khiển lời gọi ngôn ngữ thủ tục (Chương 49). Mã phiên bản 1 cũng khả chuyển hơn so với phiên bản 0, vì nó không phá vỡ các hạn chế trong thủ tục gọi hàm theo tiêu chuẩn C. Để có thêm chi tiết, hãy xem src/backend/utils/fmgr/README trong phân phối nguồn.

35.9.5. Viết mã

Trước khi chúng ta chuyển sang các chủ đề cao cấp hơn, chúng ta nên thảo luận vài qui tắc tạo mã cho các hàm ngôn ngữ C của PostgreSQL. Trong khi có thể có khả năng để tải các hàm được viết trong các ngôn ngữ khác với C trong PostgreSQL, thì điều này thường là khó (khi nó hoàn toàn có khả năng) vì các ngôn ngữ khác, như C++, FORTRAN, hoặc Pascal thường không tuân theo qui ước gọi y hệt như C. Đó là, các ngôn ngữ khác không truyền đối số và trả về các giá trị giữa các hàm theo cách y hệt đó. Vì lý do này, chúng ta sẽ giả thiết rằng các hàm ngôn ngữ C thực sự được viết trong C.

Các qui tắc cho việc viết và xây dựng các hàm C là như sau:

- Sử dụng pg_config --includedir-server để tìm ra nơi các tệp đầu đề máy chủ PostgreSQL được cài đặt trong hệ thống của bạn (hoặc hệ thống mà những người sử dụng của bạn sẽ chạy).
- Việc biên dịch và liên kết mã của bạn sao cho nó có thể được tải động vào PostgreSQL luôn đòi hỏi các cờ đặc biệt. Xem Phần 35.9.6 để có giải thích chi tiết cách làm điều đó cho hệ điều hành đặc thù của bạn.
- Hãy nhớ xác định một "khối ma thuật" cho thư viện được chia sẻ của bạn, như được mô tả

trong Phần 35.9.1.

- Khi phân phối bộ nhớ, hãy sử dụng các hàm palloc và pfree của PostgreSQL thay vì các hàm malloc và free của thư viện C tương ứng. Bộ nhớ được palloc phân bổ sẽ được giải phóng tự động ở cuối của từng giao dịch, ngăn chặn được các rò rỉ bộ nhớ.
- Luôn cho bằng zero các byte các cấu trúc của bạn bằng việc sử dụng memset (hoặc phân bổ chúng với palloc0 từ đầu). Thậm chí nếu bạn chỉ định cho từng trường cấu trúc của bạn, thì có thể sẽ có việc chêm phù hợp (các lỗ trong cấu trúc) mà chứa các giá trị rác. Không có điều này, là khó để hỗ trợ các chỉ số băm hoặc liên kết băm, khi bạn phải nhặt ra chỉ các bit có nghĩa đối với cấu trúc của bạn để tính toán băm. Trình lên kế hoạch cũng đôi khi dựa vào việc so sánh các hằng số qua đẳng thức bitwise, sao cho bạn có thể có được các kết quả lên kế hoạch không mong muốn nếu các giá trị tương đương về logic sẽ không bằng bitwise.
- Hầu hết các dạng PostgreSQL nội bộ được khai báo trong postgres.h, trong khi các giao diện của trình quản lý hàm (PG_FUNCTION_ARGS, ...) là trong fmgr.h, nên bạn sẽ cần đưa vào ít nhất 2 tệp đó. Vì các lý do của tính khả chuyển, là tốt nhất để đưa vào postgres.h trước tiên, trước bất kỳ hệ thống hoặc các tệp đầu đề nào khác của người sử dụng. Việc đưa vào postgres.h cũng sẽ bao gồm elog.h và palloc.h cho bạn.
- Các tên ký hiệu được định nghĩa bên trong các tệp đối tượng phải không xung đột với nhau hoặc với các ký hiệu được định nghĩa trong các tệp thực thi máy chủ PostgreSQL. Bạn sẽ phải đổi tên các hàm hoặc các biến nếu bạn có các thông điệp lỗi đối với hiệu ứng này.

35.9.6. Biên dịch và liên kết các hàm được tải động

Trước khi bạn có khả năng sử dụng các hàm mở rộng PostgreSQL được viết trong C của bạn, chúng phải được biên dịch và được liên kết theo một cách thức đặc biệt để tạo ra một tệp có thể được máy chủ tải tự động. Để chính xác, một *thư viện được chia (shared library)* sẻ cần phải được tạo ra.

Đối với thông tin vượt ra khỏi những gì có trong phần này thì bạn nên đọc tài liệu hệ điều hành của bạn, đặc biệt các trang chỉ dẫn cho trình biên dịch C, cc, và trình biên tập liên kết, ld. Hơn nữa, mã nguồn của PostgreSQL có vài ví dụ làm việc trong thư mục contrib. Tuy nhiên, nếu bạn dựa vào các ví dụ đó thì bạn sẽ làm cho các module của bạn phụ thuộc vào tính sẵn sàng của mã nguồn PostgreSQL.

Việc tạo ra các thư viện được chia sẻ thường là tương tự như với việc liên kết các tệp thực thi: trước hết các tệp nguồn được biên dịch thành các tệp đối tượng, sau đó các tệp đối tượng sẽ được liên kết với nhau. Các tệp đối tượng cần phải được tạo ra như là *mã độc lập vị trí* - PIC (*Position - Independent Code*), về mặt khái niệm có nghĩa là chúng có thể được đặt trong một vị trí tùy ý trong bộ nhớ khi chúng được tệp thực thỉ tải lên. (Các tệp đối tượng có ý định đối với các tệp thực thi thường không được biên dịch theo cách đó). Lệnh để liên kết một thư viện được chia sẻ chứa các cờ đặc biệt để phân biệt nó với việc liên kết một tệp thực thi (ít nhất về lý thuyết - trong vài hệ thống thì thực tế là xấu hơn nhiều).

Trong các ví dụ sau đây chúng ta giả thiết mã nguồn của bạn là trong một tệp foo.c và chúng ta sẽ tạo ra một thư viện được chia sẻ foo.so. Tệp đối tượng trung gian sẽ được gọi là foo.o trừ phi được

lưu ý khác. Thư viện được chia sẻ có thể chứa hơn một tệp đối tượng, nhưng chúng ta chỉ sử dụng 1 ở đây.

BSD/OS

Cờ của trình biên dịch để tạo ra PIC là -fpic. Cờ của trình liên kết để tạo ra các thư viện được chia sẻ là -shared.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Điều này là có khả năng áp dụng như của BSD/OS phiên bản 4.0.

FreeBSD

Cờ của trình biên dịch để tạo ra PIC là -fpic. Cờ của trình liên kết để tạo ra các thư viện được chia sẻ là -shared.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Điều này là có khả năng áp dụng như của FreeBSD phiên bản 3.0.

HP-UX

Cờ của trình biên dịch của trình biên dịch hệ thống để tạo ra PIC là +z. Khi sử dụng GCC thì nó là -fpic. Cờ của trình liên kết cho các thư viện được chia sẻ là -b. Nên:

```
cc +z -c foo.c
```

hoặc

```
gcc -fpic -c foo.c
```

và sau đó

```
ld -b -o foo.sl foo.o
```

HP-UX sử dụng phần mở rộng tệp là .sl cho các thư viện được chia sẻ, không giống như hầu hết các hệ điều hành khác.

IRIX

PIC là mặc định, không lựa chọn trình biên dịch đặc biệt nào là cần thiết. Lựa chọn của trình liên kết để tạo ra các thư viện được chia sẻ là -shared.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

Cờ của trình biên dịch để tạo PIC là -fpic. Trên một vài nền tảng trong vài tình huống -fPIC phải được sử dụng nếu -fpic không làm việc. Hãy tham chiếu tới sách chỉ dẫn GCC để có thêm thông tin. Cờ của trình biên dịch để tạo ra một thư viện được chia sẻ là -shared. Một ví dụ hoàn chỉnh trông giống như thế này:

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

MacOS X

Đây là một ví dụ. Nó giả thiết các công cụ của lập trình viên được cài đặt.

```
cc -c foo.c
```

```
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

Cờ của trình điều khiển để tạo PIC là -fpic. Đối với các hệ thống ELF, trình biên dịch với cờ -shared được sử dụng để liên kết các thư viện được chia sẻ. Trên các hệ thống phi ELF cũ hơn, ld -Bshareable được sử dụng.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

Cờ của trình biên dịch để tạo PIC là -fpic. ld -Bshareable được sử dụng để liên kết các thư viện được chia sẻ.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

Cờ của trình biên dịch để tạo PIC là -KPIC với trình biên dịch của Sun và -fpic với GCC. Để liên kết các thư viện được chia sẻ, lựa chọn của trình biên dịch là -G với hoặc trình biên dịch hoặc có khả năng lựa chọn -shared với GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

hoặc

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC là mặc định, nên lệnh biên dịch thường là một lệnh. ld với các lựa chọn đặc biệt được sử dụng để thực hiện việc liên kết.

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

Thủ tục y hệt được sử dụng với GCC thay vì trình biên dịch hệ thống; không lựa chọn đặc biệt nào được yêu cầu.

UnixWare

Cờ của trình biên dịch để tạo PIC là -K PIC với trình biên dịch của SCO và -fpic với GCC. Để liên kết các thư viện được chia sẻ, lựa chọn của trình biên dịch là -G với trình biên dịch của SCO và -shared với GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

hoặc

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Mẹo: Nếu điều này là quá phức tạp với bạn, thì bạn nên cân nhắc sử dụng GNU Libtool[1], nó dấu các khác biệt nền tảng đằng sau một giao diện thống nhất.

Tệp kết quả thư viện được chia sẻ có thể sau đó được tải vào trong PostgreSQL. Khi chỉ định tên

1 http://www.gnu.org/software/libtool/

tệp cho lệnh CREATE FUNCTION, người ta phải trao cho nó tên hoặc tệp thư viện được chia sẻ, chứ không phải tệp đối tượng trung gian. Lưu ý rằng phần mở rộng theo tiêu chuẩn của thư viện được chia sẻ của hệ thống (thường là .so hoặc .sl) có thể bị bỏ qua khỏi lệnh CREATE FUNCTION, và thường là nên bị bỏ qua vì tính khả chuyển tốt nhất.

Hãy tham chiếu ngược về Phần 35.9.1 nơi mà máy chủ kỳ vọng thấy các tập thư viện được chia sẻ.

35.9.7. Cấu trúc xây dựng mở rộng

Nếu bạn đang nghĩ về việc phân phối các module mở rộng PostgreSQL của bạn, thì việc thiết lập một hệ thống xây dựng khả chuyển cho chúng có thể là khá khó khăn. Vì thế cài đặt PostgreSQL đưa ra một hạ tầng xây dựng cho các mở rộng, gọi là PGXS, sao cho các module mở rộng đơn giản có thể được xây dựng đơn giản đối với một máy chủ được cài đặt rồi. Lưu ý là hạ tầng này không có ý định sẽ là một khung hệ thống xây dựng vạn năng mà có thể được sử dụng để xây dựng tất cả các phần mềm đang giao diện với PostgreSQL; nó đơn giản tự động hóa các qui tắc xây dựng phổ biến cho các module mở rộng máy chủ đơn giản. Để có nhiều gói phức tạp hơn, bạn cần phải viết hệ thống xây dựng của riêng bạn.

Để sử dụng hạ tầng cho mở rộng của bạn, thì bạn phải viết một tạo tệp (makefile) đơn giản. Trong tạo tệp đó, bạn cần thiết lập vài biến và cuối cùng đưa vào tạo tệp PGXS tổng thể. Đây là ví dụ xây dựng một module mở rộng có tên là isbn_issn gồm một thư viện được chia sẻ, một script SQL, và một tệp văn bản tài liệu:

```
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) –pgxs)
include $(PGXS)
```

3 dòng cuối cùng sẽ luôn là y hệt nhau. Trước đó trong tệp, bạn chỉ định các biến hoặc thêm các qui tắc làm tùy biến.

Thiết lập 1 trong 3 biến để chỉ định biến nào được xây dựng:

MODULES

liệt kê các đối tượng được chia sẻ sẽ được xây dựng từ các tệp nguồn với phần cột lõi y hệt (không bao gồm hậu tố trong danh sách này)

MODULE_big

một đối tượng được chia sẻ để xây dựng từ nhiều tệp nguồn (liệt kê các tệp đối tượng trong OBJS)

PROGRAM

một chương trình nhị phân để xây dựng (liệt kê các tệp đối tượng trong OBJS)

Các biến sau đây cũng có thể được thiết lập:

MODULEDIR

thư mục con trong đó các tệp DATA và DOCS sẽ được cài đặt (nếu không được thiết lập, thì

mặc định là contrib)

DATA

các tệp ngẫu nhiên để cài đặt trong prefix/share/$MODULEDIR

DATA_built

các tệp ngẫu nhiên để cài đặt trong prefix/share/$MODULEDIR, nó cần được xây dựng trước

DATA_TSEARCH

các tệp ngẫu nhiên để cài đặt dưới prefix/share/tsearch_data

DOCS

các tệp ngẫu nhiên để cài đặt dưới prefix/doc/$MODULEDIR

SCRIPTS

các tệp script (không phải nhị phân) để cài đặt trong prefix/bin, điều cần được xây dựng trước

SCRIPTS_built

các tệp script (không phải nhị phân) để cài đặt trong prefix/bin, điều cần được xây dựng trước

REGRESS

liệt kê các trường hợp kiểm thử hồi qui (không có hậu tố), xem bên dưới

EXTRA_CLEAN

các tệp dư thừa phải loại bỏ trong make clean

PG_CPPFLAGS

sẽ được thêm vào CPPFLAGS

PG_LIBS

sẽ được thêm vào dòng liên kết chương trình PROGRAM

SHLIB_LINK

sẽ được thêm vào dòng liên kết MODULE_big

PG_CONFIG

đường dẫn tới chương trình pg_config cho cài đặt PostgreSQL để xây dựng (thường chỉ pg_config sử dụng cái đầu tiên trong PATH của bạn)

Hãy đặt tạo tệp này như là Makefile trong thư mục nắm giữ mở rộng của bạn. Sau đó bạn có thể thực hiện make để biên dịch, và sau đó make install để cài đặt module của bạn. Mặc định, mở rộng đó được biên dịch và cài đặt cho cài đặt PostgreSQL mà tương ứng với chương trình pg_config đầu tiên được thấy trong đường dẫn của bạn. Bạn có thể sử dụng một cài đặt khác bằng việc thiết lập PG_CONFIG để trỏ tới chương trình pg_config của nó, hoặc bên trong tạo tệp hoặc ở dòng lệnh make.

Cảnh báo

Việc thay đổi PG_CONFIG chỉ làm việc khi việc xây dựng đối với PostgreSQL 8.3 hoặc sau này. Với các phiên bản cũ hơn thì nó sẽ không làm việc để thiết lập điều đó cho bất kỳ điều gì ngoại trừ pg_config; bạn phải tùy biến PATH của bạn để lựa chọn sự cài đặt để xây dựng.

Các script được liệt kê trong biến REGRESS được sử dụng cho việc kiểm thử hồi qui module của bạn, giống hệt như make installcheck được sử dụng cho máy chủ PostgreSQL chính. Để điều này làm việc thì bạn cần phải có một thư mục con có tên là sql/ trong thư mục mở rộng của bạn, trong đó bạn đặt một tệp cho từng nhóm các kiểm thử mà bạn muốn chạy. Các tệp đó nên có phần mở rộng là .sql, nó sẽ không được đưa vào trong danh sách REGRESS trong tạo tệp. Đối với từng kiểm thử sẽ có một tệp chứa kết quả được kỳ vọng trong một thư mục con có tên là expected/, với phần mở rộng là .out. Các kiểm thử đó được chạy bằng việc thực thi make installcheck, và kết quả đầu ra sẽ được so sánh với các tệp được kỳ vọng. Các khác biệt sẽ được ghi vào tệp regression.diffs ở định dạng diff -c. Lưu ý rằng việc cố chạy một kiểm thử đang bỏ qua tệp được kỳ vọng đó sẽ được báo cáo như là "lo ngại", nên hãy chắc chắn bạn có tất cả các tệp được kỳ vọng.

> **Mẹo**: Cách dễ dàng nhất để tạo các tệp được kỳ vọng là tạo các tệp rỗng, sau đó cẩn thận điều tra các tệp kết quả sau một kiểm thử được chạy (sẽ được thấy trong thư mục results/), và sao chép chúng tới expected/ nếu chúng khớp với những gì bạn muốn từ kiểm thử đó.

35.9.8. Đối số dạng tổng hợp

Các dạng tổng hợp không có một hình thức cố định giống như các cấu trúc C. Các trường hợp dạng tổng hợp có thể chứa các trường null. Hơn nữa, các dạng tổng hợp mà là một phần của một tôn ti trật tự kế thừa có thể có các trường khác nhau so với các thành viên khác của tôn ti trật tự kế thừa y hệt đó. Vì thế, PostgreSQL đưa ra một giao diện hàm cho việc truy cập các trường các dạng tổng hợp từ C.

Giả sử chúng ta muốn viết một hàm để trả lời cho truy vấn:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
        FROM emp
        WHERE name = 'Bill' OR name = 'Sam';
```

Sử dụng các quy ước gọi phiên bản 0, chúng ta có thể định nghĩa c_overpaid như:

```
#include "postgres.h"
#include "executor/executor.h" /* for GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

bool
c_overpaid(HeapTupleHeader t, /* the current row of emp */
                int32 limit)
{
        bool isnull;
        int32 salary;
        salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
        if (isnull)
                return false;
        return salary > limit;
}
```

Trong việc lập trình theo phiên bản 1, điều ở trên có thể trông giống như thế này:

```
#include "postgres.h"
#include "executor/executor.h" /* for GetAttributeByName() */
```

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
        HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);
        int32 limit = PG_GETARG_INT32(1);
        bool isnull;
        Datum salary;

        salary = GetAttributeByName(t, "salary", &isnull);
        if (isnull)
                PG_RETURN_BOOL(false);
        /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */

        PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByName là hàm hệ thống của PostgreSQL mà trả về các thuộc tính nằm ngoài hàng được chỉ định. Nó có 3 đối số: đối số dạng HeapTupleHeader được truyền vào hàm đó, tên của thuộc tính mong muốn, và tham số trả về nói liệu thuộc tính đó có là null hay không. GetAttributeByName trả về một giá trị Datum mà bạn có thể chuyển đổi thành dạng dữ liệu phù hợp bằng việc sử dụng macro phù hợp DatumGetXXX(). Lưu ý rằng giá trị trả về là vô nghĩa nếu cờ null được thiết lập; hãy luôn kiểm tra cờ null trước khi cố gắng làm bất kỳ điều gì với kết quả đó.

Cũng có GetAttributeByNum, nó lựa chọn thuộc tính đích theo số cột thay vì tên.

Lệnh sau đây khai báo hàm c_overpaid trong SQL:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
        AS 'DIRECTORY/funcs', 'c_overpaid'
        LANGUAGE C STRICT;
```

Lưu ý chúng ta đã sử dụng STRICT sao cho chúng ta đã không phải kiểm tra liệu các đối số đầu vào có từng là NULL hay không.

35.9.9. Trả về hàng (dạng tổng hợp)

Để trả về một hàng hoặc giá trị dạng tổng hợp từ một hàm ngôn ngữ C, bạn có thể sử dụng một API đặc biệt mà cung cấp các macro và các hàm để ẩn dấu hầu hết sự phức tạp của việc xây dựng các dạng dữ liệu tổng hợp. Để sử dụng API này, tệp nguồn phải bao gồm:

```
#include "funcapi.h"
```

Có 2 cách bạn có thể xây dựng một giá trị dữ liệu tổng hợp (từ nay trở đi gọi là một "bộ dữ liệu" [tuple]): bạn có thể xây dựng nó từ một mảng các giá trị dữ liệu Datum, hoặc từ một mảng các chuỗi C mà có thể được truyền tới các hàm biến đổi đầu vào của các dạng dữ liệu cột của bộ dữ liệu đó. Trong cả 2 trường hợp, trước tiên bạn cần có hoặc xây dựng một ký hiệu (descriptor) TupleDesc cho cấu trúc bộ dữ liệu đó. Khi làm việc với Datums, bạn truyền TupleDesc tới BlessTupleDesc, và sau đó gọi heap_form_tuple cho từng hàng. Khi làm việc với các chuỗi C, bạn truyền TupleDesc tới TupleDescGetAttInMetadata, và sau đó gọi BuildTupleFromCStrings cho từng hàng. Trong trường hợp

một hàm trả về một tập hợp các bộ dữ liệu, thì hãy thiết lập các bước có thể tất cả được làm một lần trong quá trình lời gọi đầu của hàm.

Vài hàm trợ giúp là sẵn sàng cho việc thiết lập TupleDesc cần thiết. Cách được khuyến cáo để làm điều này trong hầu hết các hàm trả về các giá trị tổng hợp là phải gọi:

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

truyền cấu trúc fcinfo y hệt được truyền tới bản thân việc gọi hàm đó. (Điều này tất nhiên đòi hỏi bạn sử dụng các qui ước gọi phiên bản 1). resultTypeId có thể được chỉ định như là NULL hoặc như là địa chỉ của một biến cục bộ để nhân kết quả hàm đó dạng OID. resultTupleDesc sẽ là địa chỉ của một biến TupleDesc cục bộ. Hãy kiểm tra xem kết quả có là TYPEFUNC_COMPOSITE hay không; nếu đúng, thì resultTupleDesc đã được điền bằng TupleDesc cần thiết. (Nếu không đúng, thì bạn có thể báo cáo một lỗi cùng với các dòng "hàm trả về bản ghi được gọi theo ngữ cảnh không thể chấp nhận dạng bản ghi" (function returning record called in context that cannot accept type record)).

> **Mẹo**: get_call_result_type có thể giải quyết dạng thực sự của kết quả hàm đa hình; vì thế là hữu dụng trong các hàm mà trả về các kết quả đa hình vô hướng, không chỉ các hàm trả về các tổng hợp. Đầu ra resultTypeId trước hết là hữu dụng cho các hàm trả về các vô hướng đa hình.
>
> **Lưu ý**: get_call_result_type có một người anh em get_expr_result_type, nó cũng có thể được sử dụng cho get_func_result_type, nó có thể được sử dụng chỉ khi OID hàm đó là sẵn sàng. Tuy nhiên các hàm đó không có khả năng để làm việc với các hàm được khai báo sẽ trả về record, và get_func_result_type không thể giải quyết được các dạng đa hình, vì thế bạn nên ưu tiên sử dụng get_call_result_type.

Cũ hơn, các hàm bây giờ đã lỗi thời để có TupleDescs là:

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

để có một TupleDesc cho dạng hàng của mối quan hệ được đặt tên, và:

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

để có một TupleDescs dựa vào một OID dạng. Điều này có thể được sử dụng để có một TupleDesc cho một dạng cơ bản hoặc tổng hợp. Tuy nhiên, nó sẽ không làm việc cho một hàm mà trả về record, và nó không thể giải quyết được các dạng đa hình.

Một khi bạn có một TupleDesc, hãy gọi:

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

nếu bạn có kế hoạch để làm việc với các dữ liệu, hoặc:

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

nếu bạn có kế hoạch làm việc với các chuỗi C. Nếu bạn đang viết một hàm trả về tập hợp, thì bạn có thể lưu các kết quả của các hàm đó trong cấu trúc FuncCallContext - hãy sử dụng trường tuple_desc hoặc attinmeta một cách tương ứng.

Khi làm việc với các dữ liệu Datums, hãy sử dụng:

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

để xây dựng một HeapTuple đưa ra các dữ liệu người sử dụng ở dạng các dữ liệu Datum.

Khi làm việc với các chuỗi C, hãy sử dụng:

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

để xây dựng một HeapTuple đưa ra các dữ liệu người sử dụng ở dạng chuỗi C. values là một mảng các chuỗi C, mỗi chuỗi cho từng thuộc tính của hàng trả về. Từng chuỗi C sẽ nằm ở dạng được hàm đầu vào của dạng dữ liệu thuộc tính kỳ vọng. Để trả về một giá trị null cho một trong các thuộc tính, con trỏ tương ứng trong mảng values sẽ được thiết lập về NULL. Hàm này sẽ cần được gọi một lần nữa cho từng hàng bạn trả về.

Một khi bạn đã xây dựng được một bộ dữ liệu để trả về từ hàm của bạn, nó phải được chuyển đổi thành Datum. Hãy sử dụng:

```
HeapTupleGetDatum(HeapTuple tuple)
```

để chuyển đổi một HeapTuple thành một Datum hợp lệ. Datum này có thể được trả về trực tiếp nếu bạn định trả về chỉ một hàng duy nhất, hoặc nó có thể được sử dụng như là giá trị trả về hiện hành trong một hàm được thiết lập đó trả về.

Một ví dụ có trong phần tiếp sau.

35.9.10. Trả về tập hợp

Cũng có một API đặc biệt đưa ra sự hỗ trợ cho việc trả về các tập hợp (nhiều hàng) từ một hàm ngôn ngữ C. Một hàm trả về tập hợp phải theo các qui ước gọi phiên bản 1. Hơn nữa, các tệp nguồn phải bao gồm funcapi.h, như ở trên.

Một hàm trả về tập hợp - SRF (Set-Returning Function) được gọi một lần cho từng khoản nó trả về. SRF vì thế phải lưu đủ tình trạng để nhớ những gì nó từng làm và trả về khoản tiếp theo trong từng lời gọi. Cấu trúc FuncCallContext được cung cấp để giúp kiểm soát qui trình này. Trong một hàm, fcinfo->flinfo->fn_extra được sử dụng để giữ một con trỏ tới FuncCallContext khắp các lời gọi.

```
typedef struct
{
        /*
        * Number of times we've been called before
        *
        * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
        * incremented for you every time SRF_RETURN_NEXT() is called.
        */
        uint32 call_cntr;

        /*
        * OPTIONAL maximum number of calls
        *
        * max_calls is here for convenience only and setting it is optional.
        * If not set, you must provide alternative means to know when the
        * function is done.
        */
        uint32 max_calls;

        /*
        * OPTIONAL pointer to result slot
        *
        * This is obsolete and only present for backwards compatibility, viz,
        * user-defined SRFs that use the deprecated TupleDescGetSlot().
```

```
        */
        TupleTableSlot *slot;

        /*
        * OPTIONAL pointer to miscellaneous user-provided context information
        *
        * user_fctx is for use as a pointer to your own data to retain
        * arbitrary context information between calls of your function.
        */
        void *user_fctx;

        /*
        * OPTIONAL pointer to struct containing attribute type input metadata
        *
        * attinmeta is for use when returning tuples (i.e., composite data types)
        * and is not used when returning base data types. It is only needed
        * if you intend to use BuildTupleFromCStrings() to create the return
        * tuple.
        */

        AttInMetadata *attinmeta;
        /*
        * memory context used for structures that must live for multiple calls
        *
        * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
        * by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
        * context for any memory that is to be reused across multiple calls
        * of the SRF.
        */

        MemoryContext multi_call_memory_ctx;
        /*
        * OPTIONAL pointer to struct containing tuple description
        *
        * tuple_desc is for use when returning tuples (i.e., composite data types)
        * and is only needed if you are going to build the tuples with
        * heap_form_tuple() rather than with BuildTupleFromCStrings(). Note that
        * the TupleDesc pointer stored here should usually have been run through
        * BlessTupleDesc() first.
        */
        TupleDesc tuple_desc;

}FuncCallContext;
```

Một SRF sử dụng vài hàm và macro tự động điều khiển cấu trúc FuncCallContext (và kỳ vọng thấy nó qua fn_extra). Hãy sử dụng:

SRF_IS_FIRSTCALL()

sẽ xác định liệu hàm của bạn có đang được gọi lần đầu hay lần tiếp sau. (Chỉ) trong lần gọi đầu tiên, hãy sử dụng:

SRF_FIRSTCALL_INIT()

để khởi tạo FuncCallContext. Trong từng gọi hàm, bao gồm cả lần đầu, hãy sử dụng:

SRF_PERCALL_SETUP()

để thiết lập đúng phù hợp cho việc sử dụng FuncCallContext và việc làm sạch bất kỳ dữ liệu nào còn lại trước đó từ lần truyền trước đó.

Nếu hàm của bạn có dữ liệu để trả về, hãy sử dụng:

```
SRF_RETURN_NEXT(funcctx, result)
```

để trả nó về cho người gọi. (result phải là của dạng Datum, hoặc một giá trị duy nhất hoặc bộ dữ liệu được chuẩn bị như được mô tả ở trên). Cuối cùng, khi hàm của bạn kết thúc việc trả về dữ liệu, hãy sử dụng:

```
SRF_RETURN_DONE(funcctx)
```

để làm sạch và kết thúc SRF.

Ngữ cảnh bộ nhớ mà là hiện hành khi SRF được gọi là một ngữ cảnh tạm thời mà sẽ bị xóa sạch giữa các cuộc gọi. Điều này có nghĩa là bạn không cần gọi pfree trong mỗi điều bạn đã phân bổ bằng việc sử dụng palloc; nó thế nào cũng sẽ mất đi. Tuy nhiên, nếu bạn muốn phân bổ bất kỳ dữ liệu nào cần sống sót cho tới khi SRF kết thúc chạy. Trong hầu hết các trường hợp, điều này có nghĩa là bạn sẽ chuyển sang multi_call_memory_ctx trong khi thực hiện thiết lập lời gọi đầu tiên.

Một ví dụ mã giả hoàn chỉnh trông giống như sau:

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
        FuncCallContext *funcctx;
        Datum result;
        further declarations as needed

        if (SRF_IS_FIRSTCALL())
        {
                MemoryContext oldcontext;
                funcctx = SRF_FIRSTCALL_INIT();
                oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
                /* One-time setup code appears here: */
                user code
                if returning composite
                        build TupleDesc, and perhaps AttInMetadata
                endif returning composite
                user code
                MemoryContextSwitchTo(oldcontext);
        }

        /* Each-time setup code appears here: */
        user code
        funcctx = SRF_PERCALL_SETUP();
        user code

        /* this is just one way we might test whether we are done: */
        if (funcctx->call_cntr < funcctx->max_calls)
        {
                /* Here we want to return another item: */
                user code
                obtain result Datum
                SRF_RETURN_NEXT(funcctx, result);
        }
        else
        {
                /* Here we are done returning items and just need to clean up: */
                user code
                SRF_RETURN_DONE(funcctx);
        }
}
```

Ví dụ hoàn chỉnh về một SRF đơn giản trả về một dạng tổng hợp trông giống thế này:

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
        FuncCallContext *funcctx;
        int call_cntr;
        int max_calls;
        TupleDesc tupdesc;
        AttInMetadata *attinmeta;

        /* stuff done only on the first call of the function */
        if (SRF_IS_FIRSTCALL())
        {
                MemoryContext oldcontext;

                /* create a function context for cross-call persistence */
                funcctx = SRF_FIRSTCALL_INIT();

                /* switch to memory context appropriate for multiple function calls */
                oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

                /* total number of tuples to be returned */
                funcctx->max_calls = PG_GETARG_UINT32(0);

                /* Build a tuple descriptor for our result type */
                if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
                        ereport(ERROR,
                                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                                errmsg("function returning record called in context "
                                        "that cannot accept type record")));

                /*
                * generate attribute metadata needed later to produce tuples from raw
                * C strings
                */
                attinmeta = TupleDescGetAttInMetadata(tupdesc);
                funcctx->attinmeta = attinmeta;

                MemoryContextSwitchTo(oldcontext);
        }

        /* stuff done on every call of the function */
        funcctx = SRF_PERCALL_SETUP();

        call_cntr = funcctx->call_cntr;
        max_calls = funcctx->max_calls;
        attinmeta = funcctx->attinmeta;

        if (call_cntr < max_calls) /* do when there is more left to send */
        {
                char **values;
                HeapTuple tuple;
                Datum result;

                /*
                * Prepare a values array for building the returned tuple.
                * This should be an array of C strings which will
                * be processed later by the type input functions.
                */
                values = (char **) palloc(3 * sizeof(char *));
```

```
                values[0] = (char *) palloc(16 * sizeof(char));
                values[1] = (char *) palloc(16 * sizeof(char));
                values[2] = (char *) palloc(16 * sizeof(char));

                snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
                snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
                snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

                /* build a tuple */
                tuple = BuildTupleFromCStrings(attinmeta, values);

                /* make the tuple into a datum */
                result = HeapTupleGetDatum(tuple);

                /* clean up (this is not really necessary) */
                pfree(values[0]);
                pfree(values[1]);
                pfree(values[2]);
                pfree(values);

                SRF_RETURN_NEXT(funcctx, result);
        }
        else /* do when there is no more left */
        {
                SRF_RETURN_DONE(funcctx);
        }
}
```

Một cách để khai báo hàm này trong SQL là:

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
        RETURNS SETOF __retcomposite
        AS 'filename', 'retcomposite'
        LANGUAGE C IMMUTABLE STRICT;
```

Một cách khác là sử dụng các tham số OUT:

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
        OUT f1 integer, OUT f2 integer, OUT f3 integer)
        RETURNS SETOF record
        AS 'filename', 'retcomposite'
        LANGUAGE C IMMUTABLE STRICT;
```

Lưu ý là trong phương pháp này dạng đầu ra của hàm một cách chính thức là dạng record nặc danh.

Thư mục contrib/tablefunc trong phân phối nguồn chứa nhiều ví dụ hơn các hàm trả về tập hợp.

35.9.11. Đối số đa hình và dạng trả về

Các hàm ngôn ngữ C có thể được khai báo để chấp nhận và trả về các dạng đa hình anyelement, anyarray, anynonarray, và anyenum. Xem Phần 35.2.5 để có giải thích chi tiết hơn về các hàm đa hình. Khi các đối số hàm hoặc các dạng trả về được xác định như là các dạng đa hình, thì tác giả hàm không thể biết trước dạng dữ liệu nào nó sẽ được gọi cùng, hoặc cần phải trả về. Có 2 thủ tục được đưa ra trong fmgr.h để cho phép một hàm C phiên bản 1 phát hiện ra các dạng dữ liệu thực sự của các đối số của nó và dạng mà nó được kỳ vọng trả về. Các thủ tục đó được gọi là

get_fn_expr_rettype(FmgrInfo *flinfo) và get_fn_expr_argtype(FmgrInfo *flinfo, int argnum). Chúng trả về kết quả hoặc đối số OID dạng, hoặc InvalidOid nếu thông tin là không có sẵn. Cấu trúc flinfo thường được truy cập như fcinfo->flinfo. Tham số argnum là dựa vào zero. get_call_result_type cũng có thể được sử dụng như một lựa chọn thay thế cho get_fn_expr_rettype.

Ví dụ, giả sử chúng ta muốn viết một hàm để chấp nhận một phần tử duy nhất của bất kỳ dạng nào, và trả về một mảng 1 chiều của dạng đó:

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
        ArrayType *result;
        Oid element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
        Datum element;
        bool isnull;
        int16 typlen;
        bool typbyval;
        char typalign;
        int ndims;
        int dims[MAXDIM];
        int lbs[MAXDIM];

        if (!OidIsValid(element_type))
                elog(ERROR, "could not determine data type of input");

        /* get the provided element, being careful in case it's NULL */
        isnull = PG_ARGISNULL(0);
        if (isnull)
                element = (Datum) 0;
        else
                element = PG_GETARG_DATUM(0);

        /* we have one dimension */
        ndims = 1;
        /* and one element */
        dims[0] = 1;
        /* and lower bound is 1 */
        lbs[0] = 1;

        /* get required info about the element type */
        get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

        /* now build the array */
        result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                    element_type, typlen, typbyval, typalign);

        PG_RETURN_ARRAYTYPE_P(result);
}
```

Lệnh sau đây khai báo hàm make_array trong SQL:

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
        AS 'DIRECTORY/funcs', 'make_array'
        LANGUAGE C IMMUTABLE;
```

Có một phương án đa hình mà chỉ sẵn sàng cho các hàm ngôn ngữ C: chúng có thể được khai báo để lấy các tham số dạng "any". (Lưu ý rằng tên dạng này phải được nằm trong các dấu ngoặc kép, vì nó cũng là một từ được để dành của SQL). Điều này làm việc giống như anyelement ngoại trừ là nó

không ràng buộc các đối số bất kỳ nào "any" khác sẽ là dạng y hệt, chúng cũng không giúp xác định dạng kết quả của hàm đó. Một hàm ngôn ngữ C cũng có thể khai báo tham số cuối cùng của nó sẽ là VARIADIC "any". Điều này sẽ khớp với một hoặc nhiều đối số thực hơn của bất kỳ dạng nào (không nhất thiết là dạng y hệt). Các đối số đó sẽ không được thu thập trong một mảng như xảy ra với các hàm biến thiên thông thường; chúng sẽ chỉ được truyền tới hàm đó một cách riêng rẽ. Macro PG_NARGS() và các phương pháp được mô tả ở trên phải được sử dụng để xác định số các đối số thực và các dạng của chúng khi sử dụng tính năng này.

35.9.12. Bộ nhớ được chia sẻ và LWLocks

Các phần bổ sung thêm (Add-ins) có thể để dành LWLocks và sự phân bổ bộ nhớ được chia sẻ khi khởi động máy chủ. Thư viện được chỉ sẻ của phần bổ sung thêm này phải được tải lên trước bằng việc chỉ định nó trong shared_preload_libraries. Bộ nhớ được chia sẻ được gọi để dành:

```
void RequestAddinShmemSpace(int size)
```

từ hàm _PG_init của bạn.

LWLocks được gọi để dành:

```
LWLocks are reserved by calling:
void RequestAddinLWLocks(int n)
```

từ _PG_init.

Để tránh sự đuổi theo các điều kiện có khả năng xảy ra, từng phần phụ trợ (backend) nên sử dụng LWLock AddinShmemInitLock khi kết nối với và khởi tạo sự phân bổ bộ nhớ chia sẻ của nó, như:

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->mylockid = LWLockAssign();
        }
        LWLockRelease(AddinShmemInitLock);
}
```

35.10. Tổng hợp do người sử dụng định nghĩa

Các hàm tổng hợp (tính tổng) trong PostgreSQL được trình bày có lưu ý về các *hàm biến đổi tình trạng* và các *giá trị tình trạng*. Đó là, các vận hành tính tổng bằng việc sử dụng giá trị tình trạng được cập nhật như từng hàng đầu vào liên tục được xử lý. Để xác định một hàm tính tổng mới, người ta chọn một dạng dữ liệu cho giá trị tình trạng đó, một giá trị ban đầu cho tình trạng đó, và một hàm biến đổi tình trạng. Hàm biến đối tình trạng là hệt như một hàm thông thường mà cũng có thể được sử dụng ngoài ngữ cảnh của sự tính tổng. *Hàm cuối cùng* cũng có thể được chỉ định, trong trường hợp kết quả mong muốn của sự tính tổng là khác với dữ liệu cần phải được giữ trong giá trị

tình trạng đang chạy.

Vì thế, bổ sung thêm vào đối số và các dạng dữ liệu kết quả được một người sử dụng sự tính tổng đó thấy, có dạng dữ liệu giá trị tình trạng nội bộ có thể là khác với cả đối số và các dạng kết quả.

Nếu chúng ta xác định sự tính tổng không sử dụng hàm cuối cùng, thì chúng ta có sự tính tổng tính toán một hàm đang chạy các giá trị cột từ mỗi hàng. sum là một ví dụ về dạng tính tổng này. sum bắt đầu ở zero và luôn thêm giá trị hàng hiện hành vào tổng đang chạy của nó. Ví dụ, nếu chúng ta muốn tính tổng sum để làm việc trong một dạng dữ liệu cho các số phức tạp, thì chúng ta chỉ cần hàm bổ sung thêm cho dạng dữ liệu đó. Định nghĩa tính tổng đó có thể là:

```
CREATE AGGREGATE sum (complex)
(
        sfunc = complex_add,
        stype = complex,
        initcond = '(0,0)'
);

SELECT sum(a) FROM test_complex;

   sum
-------------
(34,53.9)
```

(Lưu ý là chúng ta đang dựa vào việc quá tải hàm: có nhiều hơn một sự tính tổng có tên là sum, nhưng PostgreSQL có thể chỉ ra dạng tính tổng - sum nào áp dụng được cho cột dạng complex).

Định nghĩa ở trên của sum sẽ trả về zero (điều kiện tình trạng ban đầu) nếu không có các giá trị đầu vào không null. Có lẽ chúng ta muốn trả về null trong trường hợp đó - tiêu chuẩn SQL kỳ vọng sum sẽ hành xử theo cách đó. Chúng ta có thể làm điều này đơn giản bằng việc bỏ qua cụm từ initcond, sao cho điều kiện tình trạng ban đầu là null. Thông thường thì điều này có thể có nghĩa là sfunc có thể cần phải kiểm tra đầu vào điều kiện tình trạng null. Nhưng đối với sum và vài sự tính tổng đơn giản khác như max và min, là đủ để chèn giá trị đầu vào không null đầu tiên vào biến tình trạng và sau đó bắt đầu áp dụng hàm biến đổi ở giá trị đầu vào không null thứ 2. PostgreSQL sẽ làm điều đó tự động nếu điều kiện ban đầu là null và hàm biến đổi được đánh dấu là "khắt khe" (strict) (nghĩa là, sẽ không được gọi cho các đầu vào null).

Một bit khác của hành vi mặc định cho một hàm biến đổi "khắt khe" là giá trị tình trạng trước đó được giữ lại không đổi bất kỳ khi nào một giá trị đầu vào null xảy ra. Vì thế, các giá trị null sẽ bị bỏ qua. Nếu bạn cần vài hành vi khác cho các đầu vào null, hãy không khai báo hàm biến đổi đó như là khắt khe; thay vào đó hãy mã nó để kiểm thử các đầu vào null và làm bất kỳ điều gì cần thiết.

avg - trung bình (average) là một ví dụ phức tạp hơn của sự tính tổng. Nó đòi hỏi 2 mẩu tình trạng đang chạy: tổng của các đầu vào và sự tính toán số các đầu vào. Kết quả cuối cùng có được bằng việc chia các lượng đó. Trung bình thường được triển khai bằng việc sử dụng một mảng như là giá trị tình trạng. Ví dụ, triển khai avg(float8) được xây dựng sẵn trông giống như:

```
CREATE AGGREGATE avg (float8)
(
        sfunc = float8_accum,
        stype = float8[],
        finalfunc = float8_avg,
```

```
        initcond = '{0,0,0}'
);
```

(float8_accum đòi hỏi một mảng 3 phần tử, không chỉ 2 phần tử, vì nó cộng dồn tổng bình phương cũng như tổng tính đếm các đầu vào. Điều này vì thế có thể được sử dụng cho một vài tính tổng khác ngoài avg).

Các hàm tính tổng có thể sử dụng các hàm biến đổi tình trạng đa hình hoặc các hàm cuối cùng, sao cho các hàm y hệt có thể được sử dụng để triển khai nhiều tính tổng. Xem Phần 35.2.5 để có sự giải thích về các hàm đa hình. Đi một bước tiếp, bản thân hàm tính tổng có thể được chỉ định với (các) dạng đầu vào đa hình và dạng tình trạng, cho phép một định nghĩa tính tổng duy nhất để phục vụ cho nhiều dạng dữ liệu đầu vào. Đây là một ví dụ tính tổng đa hình:

```
CREATE AGGREGATE array_accum (anyelement)
(
        sfunc = array_append,
        stype = anyarray,
        initcond = '{}'
);
```

Ở đây, dạng tình trạng thực sự cho bất kỳ lời gọi tính tổng nào là dạng mảng có dạng đầu vào thực sự như các phần tử. Hành vi của tính tổng là để nối tất cả các đầu vào vào trong một mảng dạng đó. (Lưu ý: tính tổng được xây dựng sẵn array_agg đưa ra chức năng tương tự, với hiệu năng tốt hơn so với định nghĩa này có thể có).

Đây là đầu ra có sử dụng 2 dạng dữ liệu thực sự khác nhau như là các đối số:

```
SELECT attrelid::regclass, array_accum(attname)
        FROM pg_attribute
        WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
        GROUP BY attrelid;

attrelid          | array_accum
--------------------+---------------------------------------------------------
pg_tablespace | {spcname,spcowner,spclocation,spcacl}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
        FROM pg_attribute
        WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
        GROUP BY attrelid;

attrelid          | array_accum
--------------------+--------------------------
pg_tablespace | {name,oid,text,aclitem[]}
(1 row)
```

Một hàm được viết trong C có thể dò tìm ra nó đang được gọi như một hàm cuối cùng hoặc biến đổi tính tổng bằng việc gọi AggCheckCallContext, ví dụ:

```
if (AggCheckCallContext(fcinfo, NULL))
```

Một lý do cho việc kiểm tra điều này là khi nó là đúng cho một hàm biến đổi, thì đầu vào đầu tiên phải là một giá trị biến đổi tạm thời và vì thế có thể an toàn được sửa đổi tại chỗ thay vì việc phân

bổ một bản sao mới. Xem int8inc() để có một ví dụ. (Đây là trường hợp duy nhất nơi mà là an toàn cho một hàm để sửa đổi một đầu vào truyền bằng tham chiếu. Đặc biệt, các hàm cuối cùng tính tổng sẽ không sửa đổi các đầu vào của chúng trong bất kỳ trường hợp nào, vì trong một vài trường hợp chúng sẽ được tái thực thi trong giá trị biến đổi cuối cùng y hệt).

Để có thêm chi tiết, hãy xem lệnh CREATE AGGREGATE.

35.11. Dạng do người sử dụng định nghĩa

Như được mô tả ở Phần 35.2, PostgreSQL có thể được mở rộng để hỗ trợ các dạng dữ liệu mới. Phần này mô tả cách để xác định các dạng cơ bản mới, là các dạng dữ liệu được định nghĩa dưới mức ngôn ngữ SQL. Việc tạo một dạng cơ bản mới đòi hỏi việc triển khai các hàm để vận hành ở dạng trong một ngôn ngữ mức thấp, thường là C.

Các ví dụ trong phần này có thể thấy được trong complex.sql và complex.c trong thư mục src/tutorial của phân phối nguồn. Xem tệp README (đọc tôi) trong thư mục đó để có các chỉ dẫn về việc chạy các ví dụ đó.

Một dạng do người sử dụng định nghĩa phải luôn có các hàm đầu vào và đầu ra. Các hàm đó xác định cách dạng đó xuất hiện trong các chuỗi (cho đầu vào của người sử dụng và đầu ra cho người sử dụng đó) và cách mà dạng đó được tổ chức trong bộ nhớ. Hàm đầu vào lấy một chuỗi ký tự kết thúc bằng null như là đối số của nó và trả về trình bày nội bộ (trong bộ nhớ) của dạng đó. Hàm đầu ra lấy trình bày nội bộ dạng đó như là đối số và trả về một chuỗi ký tự kết thúc bằng null. Nếu chúng ta muốn làm bất kỳ điều gì hơn với dạng đó so với chỉ lưu trữ nó, thì chúng ta phải cung cấp các hàm bổ sung thêm để triển khai bất kỳ hoạt động nào chúng ta muốn có cho dạng đó.

Giả sử chúng ta muốn định nghĩa một dạng complex mà thể hiện các số phức tạp. Một cách tự nhiên để thể hiện một số phức tạp trong bộ nhớ có thể là cấu trúc C sau đây:

```
typedef struct Complex {
        double x;
        double y;
} Complex;
```

Chúng ta sẽ cần làm điều này thành dạng truyền bằng tham chiếu, khi mà quá lớn để vừa trong một giá trị Datum duy nhất.

Như sự trình bày chuỗi bên ngoài của dạng đó, chúng ta chọn một chuỗi dạng (x,y).

Các hàm đầu vào và đầu ra thường không khó để viết, đặc biệt hàm đầu ra. Nhưng khi xác định trình bày chuỗi dạng đó, hãy nhớ là bạn cuối cùng phải viết một trình phân tích cú pháp mạnh mẽ cho sự trình bày đó như hàm đầu vào của bạn. Ví dụ:

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
        char *str = PG_GETARG_CSTRING(0);
        double x,
```

```
                y;
        Complex *result;

        if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
                ereport(ERROR,
                        (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                        errmsg("invalid input syntax for complex: \"%s\"",
                                str)));

        result = (Complex *) palloc(sizeof(Complex));
        result->x = x;
        result->y = y;
        PG_RETURN_POINTER(result);
}
```

Hàm đầu ra có thể đơn giản là:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
        Complex *complex = (Complex *) PG_GETARG_POINTER(0);
        char *result;

        result = (char *) palloc(100);
        snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
        PG_RETURN_CSTRING(result);
}
```

Bạn nên cẩn thận để làm các hàm đầu vào và đầu ra nghịch đảo với nhau. Nếu bạn không làm, bạn sẽ có các vấn đề nghiêm trọng khi bạn cần đổ dữ liệu của bạn vào một tệp và sau đó đọc nó trở lại. Đây là vấn đề đặc biệt phổ biến khi các số dấu chấm động có liên quan.

Như một sự lựa chọn, một dạng do người sử dụng định nghĩa có thể đưa ra các thủ tục đầu vào và đầu ra nhị phân. I/O nhị phân thường là nhanh hơn nhưng ít khả chuyển hơn so với I/O văn bản thô. Như với I/O văn bản thô, tùy bạn định nghĩa chính xác những gì trình bày nhị phân bên ngoài là. Hầu hết các dạng dữ liệu được xây dựng sẵn cố gắng cung cấp một trình bày nhị phân độc lập với máy. Đối với complex, chúng ta sẽ đưa trở lại trong các bộ chuyển đổi I/O nhị phân cho dạng float8:

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
        StringInfo buf = (StringInfo) PG_GETARG_POINTER(0);
        Complex *result;

        result = (Complex *) palloc(sizeof(Complex));
        result->x = pq_getmsgfloat8(buf);
        result->y = pq_getmsgfloat8(buf);
        PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
```

```
complex_send(PG_FUNCTION_ARGS)
{
        Complex *complex = (Complex *) PG_GETARG_POINTER(0);
        StringInfoData buf;

        pq_begintypsend(&buf);
        pq_sendfloat8(&buf, complex->x);
        pq_sendfloat8(&buf, complex->y);
        PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Một khi chúng ta đã viết các hàm I/O và đã biên dịch chúng thành một tệp nhị phân được chia sẻ, thì chúng ta có thể định nghĩa dạng complex trong SQL. Trước hết chúng ta khai báo nó như một dạng trình biên dịch shell:

```
CREATE TYPE complex;
```

Điều này phục vụ như một chỗ để chừa ra mà cho phép chúng ta tham chiếu tới dạng đó trong khi định nghĩa các hàm I/O của nó. Bây giờ chúng ta định nghĩa các hàm I/O:

```
CREATE FUNCTION complex_in(cstring)
        RETURNS complex
        AS 'filename'
        LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
        RETURNS cstring
        AS 'filename'
        LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
        RETURNS complex
        AS 'filename'
        LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
        RETURNS bytea
        AS 'filename'
        LANGUAGE C IMMUTABLE STRICT;
```

Cuối cùng, chúng ta có thể đưa ra định nghĩa đầy đủ dạng dữ liệu:

```
CREATE TYPE complex (
        internallength = 16,
        input = complex_in,
        output = complex_out,
        receive = complex_recv,
        send = complex_send,
        alignment = double
);
```

Khi bạn định nghĩa một dạng cơ bản mới, thì PostgreSQL tự động đưa ra sự hỗ trợ cho các mảng dạng đó. Dạng mảng thường có tên y hệt như dạng cơ bản với ký tự gạch chân (_) làm tiền tố.

Một khi dạng dữ liệu đó tồn tại, thì chúng ta có thể khai báo các hàm bổ sung để đưa ra các hành động có ích ở dạng dữ liệu đó. Các toán tử có thể sau đó được định nghĩa bên trên các hàm đó, và nếu cần, các lớp toán tử có thể được tạo ra để hỗ trợ cho việc đánh chỉ số dạng dữ liệu đó. Các lớp

bổ sung thêm sẽ được thảo luận trong các phần sau.

Nếu các giá trị dạng dữ liệu của bạn khác nhau về kích cỡ (ở dạng nội bộ), thì bạn nên tạo ra dạng dữ liệu TOASTable (xem Phần 54.2). Bạn nên làm điều này thậm chí nếu dữ liệu đó luôn quá nhỏ để được nén hoặc lưu trữ bên ngoài, vì TOAST cũng có thể tiết kiệm không gian trong các dữ liệu nhỏ, bằng việc giảm tổng chi phí các đầu đề.

Để làm điều này, trình bày nội bộ phải tuân theo hình thức tiêu chuẩn cho dữ liệu độ dài biến đổi: 4 byte đầu phải là một trường char[4] mà nó không bao giờ được truy cập trực tiếp (thường được gọi là vl_len_). Bạn phải sử dụng SET_VARSIZE() để lưu trữ kích cỡ của datum trong trường này và VARSIZE() để truy xuất nó. Việc vận hành các hàm C trong dạng dữ liệu đó phải luôn thận trọng để mở bất kỳ giá trị được nướng (toasted) nào mà chúng được giữ, bằng việc sử dụng PG_DETOAST_DATUM. Sau đó, khi chạy lệnh CREATE TYPE, hãy chỉ định độ dài nội bộ như là variable và chọn lựa chọn lưu trữ phù hợp.

Nếu sự điều chỉnh là không quan trọng (hoặc chỉ cho một hàm đặc biệt hoặc vì dạng dữ liệu chỉ định sự điều chỉnh byte bất kỳ cách gì) thì có thể tránh vài tổng chi phí của PG_DETOAST_DATUM. Thay vào đó, bạn có thể sử dụng PG_DETOAST_DATUM_PACKED (thường là ẩn bằng việc xác định macro GETARG_DATATYPE_PP) và sử dụng các macro VARSIZE_ANY_EXHDR và VARDATA_ANY để truy cập một datum được đóng gói tiềm tàng. Một lần nữa, dữ liệu được các macro đó trả về không được điều chỉnh thậm chí nếu định nghĩa dạng dữ liệu đó chỉ định một sự điều chỉnh. Nếu sự điều chỉnh đó là quan trọng thì bạn phải đi qua giao diện phổ dụng PG_DETOAST_DATUM.

> **Lưu ý**: Mã cũ hơn thường khai báo vl_len_ như một trường int32 thay vì char[4]. Điều này là OK miễn là định nghĩa cấu trúc có các trường khác mà có ít nhất điều chỉnh int32. Nhưng là nguy hiểm để sử dụng một định nghĩa cấu trúc như vậy khi làm việc với một datum không được điều chỉnh tiềm tàng; trình biên dịch có thể lấy nó như là giấy phép để giả thiết datum đó thực sự được điều chỉnh, dẫn tới làm nản chí cốt lõi trong các kiến trúc là khắt khe đối với sự điều chỉnh.

Để có thêm chi tiết, hãy xem mô tả về lệnh CREATE TYPE.

35.12. Toán tử do người sử dụng định nghĩa

Mỗi toán tử là "viên đường cú pháp" cho một lời gọi tới một hàm bên dưới mà thực hiện công việc thực; nên bạn trước tiên phải tạo hàm bên dưới đó trước khi bạn có thể tạo toán tử đó. Tuy nhiên, một toán tử *không chỉ là* viên đường cú pháp, vì nó mang thông tin bổ sung giúp trình lên kế hoạch truy vấn tối ưu hóa các truy vấn sử dụng toán tử đó. Phần tiếp theo sẽ dành để giải thích thông tin bổ sung thêm đó.

PostgreSQL hỗ trợ toán tử đặc biệt unary trái, unary phải, và các toán tử nhị phân. Các toán tử có thể bị quá tải; đó là, tên toán tử y hệt có thể được sử dụng cho các toán tử khác nhau có các số và các dạng toán hạng khác nhau. Khi một truy vấn được thực thi, hệ thống xác định toán tử đó gọi từ số và các dạng toán hạng được cung cấp.

Đây là một ví dụ của việc tạo một toán tử để thêm 2 số phức tạp. Chúng ta giả thiết chúng ta đã tạo ra rồi định nghĩa dạng complex (xem Phần 35.11). Trước hết chúng ta cần một hàm thực hiện công việc đó, sau đó chúng ta có thể định nghĩa toán tử đó:

```
CREATE FUNCTION complex_add(complex, complex)
        RETURNS complex
        AS 'filename', 'complex_add'
        LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
        leftarg = complex,
        rightarg = complex,
        procedure = complex_add,
        commutator = +
);
```

Bây giờ chúng ta có thể thực thi một truy vấn như thế này:

```
SELECT (a + b) AS c FROM test_complex;

        c
----------------------
(5.2,6.05)
(133.42,144.95)
```

35.13. Toán tử thông tin tối ưu hóa

Định nghĩa toán tử của PostgreSQL có thể bao gồm vài mệnh đề tùy chọn nói cho hệ thống những điều hữu ích về cách toán tử đó hành xử. Các mệnh đề đó sẽ được đưa ra bất kỳ khi nào phù hợp, vì chúng có thể tăng tốc đáng kể trong thực thi các truy vấn sử dụng toán tử đó. Nhưng nếu bạn đưa chúng ra, thì bạn phải chắc chắn rằng chúng là đúng! Sử dụng không đúng một mệnh đề tối ưu hóa có thể làm cho các truy vấn bị chậm, đầu ra hơi sai, hoặc những điều tồi tệ khác. Bạn luôn có thể để ra ngoài mệnh đề tối ưu hóa nếu bạn không chắc về nó; hậu quả duy nhất là các truy vấn có thể chạy chậm hơn so với chúng cần phải.

Các mệnh đề tối ưu hóa bổ sung thêm có thể được thêm vào trong các phiên bản PostgreSQL trong tương lai. Các mệnh đề được mô tả ở đây tất cả là các mệnh đề mà phiên bản 9.0.13 hiểu được.

35.13.1. COMMUTATOR

Mệnh đề COMMUTATOR, nếu được đưa ra, đặt tên cho một toán tử là hoán tử của toán tử đang được định nghĩa. Chúng ta nói rằng toán tử A là hoán tử của toán tử B nếu (x A y) bằng (y B x) với tất cả các giá trị đầu vào x, y có khả năng. Lưu ý là B cũng là hoán tử của A. Ví dụ, các toán tử < và > cho dạng dữ liệu đặt biệt và thường giao hoán với nhau, và toán tử + thường giao hoán với bản thân nó. Nhưng toán tử - thường không là giao hoán với bất kỳ điều gì.

Dạng toán hạng bên trái của một toán tử có thể hoán đổi là y hệt như dạng toán hạng bên phải của hoán tử của nó, và ngược lại. Vì thế tên của toán hạng của hoán tử là tất cả điều mà PostgreSQL cần để tra cứu hoán tử đó, và đó là tất cả điều cần thiết được đưa ra trong mệnh đề COMMUTATOR.

Là sống còn để đưa ra thông tin hoán tử cho các toán tử sẽ được sử dụng trong các chỉ số và các mệnh đề liên kết, vì điều này cho phép trình tối ưu hóa truy vấn "lật quanh" một mệnh đề như vậy

đối với các mẫu cần thiết cho các dạng kế hoạch khác nhau. Ví dụ, hãy xem xét một truy vấn với một mệnh đề WHERE giống như `tab1.x = tab2.y`, trong đó `tab1.x` và `tab2.y` là một dạng do người sử dụng định nghĩa, và giả sử rằng `tab2.y` được đánh chỉ số. Trình tối ưu hóa không thể sinh ra sự quét chỉ số trừ phi nó có thể xác định cách để lật mệnh đề đó quanh về `tab2.y = tab1.x`, vì máy quét chỉ số đó kỳ vọng thấy cột được đánh chỉ số ở bên trái của toán tử mà nó được đưa ra. PostgreSQL sẽ không đơn giản giả thiết rằng đây là một sự biến đổi hợp lệ - người sáng tạo ra toán tử = phải chỉ định rằng nó là hợp lệ, bằng việc đánh dấu toán tử đó với thông tin của hoán tử.

Khi bạn đang xác định một toán tử tự hoán đổi, bạn chỉ việc thực hiện nó. Khi bạn đang xác định một cặp các toán tử có thể hoán đổi, những điều đó là mẹo mực một chút: làm thế nào toán tử thứ nhất sẽ được định nghĩa tham chiếu tới toán tử khác, điều mà bạn còn chưa xác định được? Có 2 giải pháp cho vấn đề này:

- Một cách là bỏ qua mệnh đề COMMUTATOR trong toán tử thứ nhất mà bạn định nghĩa, và sau đó đưa ra một mệnh đề trong định nghĩa toán tử thứ 2. Vì PostgreSQL biết rằng các toán tử thay thế đi theo đôi, khi nó thấy định nghĩa thứ 2 thì nó sẽ tự động quay ngược lại và điền vào mệnh đề COMMUTATOR còn thiếu trong định nghĩa thứ nhất.
- Cách khác, cách thẳng trực tiếp hơn, là đưa các mệnh đề COMMUTATOR vào cả 2 định nghĩa đó. Khi PostgreSQL xử lý định nghĩa thứ nhất và nhận thức được COMMUTATOR tham chiếu tới một toán tử chưa tồn tại, thì hệ thống sẽ làm một đầu vào giả cho toán tử đó trong catalog hệ thống. Đầu vào giả này sẽ có dữ liệu hợp lệ chỉ cho tên của toán tử đó, các dạng toán hạng trái và phải, và dạng kết quả, vì đó là tất cả điều mà PostgreSQL có thể suy luận ra ở thời điểm này. Đầu vào catalog toán tử thứ nhất sẽ liên kết tới đầu vào giả với thông tin bổ sung thêm từ định nghĩa thứ 2. Nếu bạn cố sử dụng toán tử giả trước khi nó được điền vào, thì bạn chỉ có được một thông báo lỗi.

35.13.2. NEGATOR

Mệnh đề NEGATOR, nếu được đưa ra, đặt tên cho toán tử là sự phủ định của toán tử đang được định nghĩa. Chúng ta nói rằng toán tử A là phủ định của toán tử B nếu cả 2 trả về các kết quả Boolean và (x A y) không (NOT) bằng (x B y) đối với tất cả các đầu vào có khả năng của x, y. Lưu ý là B cũng là phủ định của A. Ví dụ, < và >= là một cặp phủ định đối với hầu hết các dạng dữ liệu. Một toán tử có thể không bao giờ hợp lệ là phủ định của chính nó.

Không giống như các hoán tử, một cặp các toán tử unary có thể được đánh dấu hợp lệ cho từng hoán tử; điều đó có nghĩa là (A x) không (NOT) bằng (B x) với tất cả x, hoặc tương đương với các toán tử unary bên phải.

Một phủ định toán tử phải có cùng y hệt các dạng toán hạng bên trái và/hoặc bên phải khi toán tử đó được định nghĩa, sao cho hệt như với COMMUTATOR, chỉ tên toán tử đó cần được đưa ra trong mệnh đề NEGATOR.

Việc đưa ra một phủ định là rất hữu dụng cho trình tối ưu hóa truy vấn vì nó cho phép các đẳng thức như NOT (x = y) sẽ được đơn giản hóa thành x < > y. Điều này gặp thường xuyên hơn bạn nghĩ, vì các phép toán NOT có thể được chèn vào như một sự tuần tự của các dàn xếp lại khác.

Các cặp toán tử phủ định có thể được định nghĩa bằng việc sử dụng các phương pháp y hệt được giải thích ở trên cho các cặp hoán tử.

35.13.3. RESTRICT

Mệnh đề RESTRICT, nếu được đưa ra, đặt tên cho một hàm ước lượng có chọn lọc giới hạn đối với toán tử. (Lưu ý là đây là tên của hàm, không phải là tên toán tử). Các mệnh đề RESTRICT chỉ có ý nghĩa cho các toán tử nhị phân trả về boolean. Ý tưởng đằng sau một ước lượng có chọn lọc giới hạn là để phỏng đoán một phần các hàng trong một bảng sẽ thỏa mãn một điều kiện của mệnh đề WHERE dạng:

```
column OP constant
```

cho toán tử hiện hành và giá trị hằng số đặc biệt. Điều này trợ giúp cho trình tối ưu hóa bằng việc trao cho nó vài ý tưởng về có bao nhiêu hàng sẽ bị các mệnh đề WHERE giới hạn mà có dạng này. (Điều gì xảy ra nếu hằng số đó là ở bên trái, bạn có thể đoán chứ? Vâng, đó là một trong những điều mà COMMUTATOR …)

Việc viết các hàm ước lượng có chọn lọc giới hạn vượt ra ngoài phạm vi của chương này, nhưng may thay bạn có thể thường chỉ sử dụng một trong số các ước lượng tiêu chuẩn của hệ thống cho nhiều toán tử của riêng bạn. Chúng là các ước lượng giới hạn tiêu chuẩn:

```
eqsel for =
neqsel for <>
scalarltsel for < or <=
scalargtsel for > or >=
```

Có lẽ dường như là hơi kỳ lạ rằng chúng là các chủng loại, nhưng chúng có ý nghĩa nếu bạn nghĩ về điều đó. Dấu = thường sẽ chỉ chấp nhận một phần nhỏ các hàng trong một bảng; < > thường sẽ từ chối chỉ một phần nhỏ. Dầu < sẽ chấp nhận một phần phụ thuộc vào đâu là nơi hằng số được đưa ra nằm trong dải các giá trị cho cột của bảng đó (điều mà, chỉ xảy ra như vậy, là thông tin được ANALYZE thu thập và được làm cho sẵn sàng cho ước lượng có chọn lọc). <= sẽ chấp nhận phần lớn hơn một chút so với < cho hằng số so sánh y hệt, nhưng chúng đủ gần để không đáng phân biệt, đặc biệt khi chúng ta có lẽ không làm tốt hơn được so với một sự phỏng đoán thô bằng bất kỳ cách gì. Các lưu ý tương tự dành cho > và >=.

Bạn có thể thường bỏ qua việc sử dụng hoặc eqsel hoặc neqsel cho các toán tử có sự lựa chọn rất cao hoặc rất thấp, thậm chí nếu chúng không thực sự bằng nhau hoặc không bằng nhau. Ví dụ, các toán tử hình học gần bằng nhau sử dụng eqsel để giả thiết là chúng thường sẽ chỉ khớp nhau một phần nhỏ các khoản đầu vào trong một bảng.

Bạn có thể sử dụng scalarltsel và scalargtsel để so sánh các dạng dữ liệu có vài nghĩa nhạy cảm đang được chuyển đổi thành các lượng vô hướng số cho các so sánh dãy. Nếu có khả năng, hãy thêm dạng dữ liệu cho các dạng được hàm convert_to_scalar() hiểu trong src/backend/utils/adt/selfuncs.c. (Rốt cuộc, hàm này sẽ bị thay thế bằng các hàm dạng theo dữ liệu được xác định qua một cột của catalog hệ thống pg_type; nhưng điều đó còn chưa xảy ra). Nếu bạn không làm điều này, thì mọi điều vẫn sẽ làm việc, nhưng các ước lượng của trình tối ưu hóa sẽ không được tốt như có thể.

Có các hàm ước lượng có chọn lọc bổ sung được thiết kế cho các toán tử hình học trong src/backend/utils/adt/geo_selfuncs.c: areasel, positionsel, và contsel. Khi tài liệu này được viết thì chúng mới có một chút, nhưng bạn có thể muốn sử dụng chúng (hoặc thậm chí tốt hơn, cải thiện chúng) bằng mọi cách.

35.13.4. JOIN

Mệnh đề JOIN, nếu được đưa ra, đặt tên cho một hàm ước lượng có chọn lọc liên kết đối với toán tử. (Lưu ý đây là tên hàm, không phải tên toán tử). Các mệnh đề JOIN chỉ có ý nghĩa cho các toán tử nhị phân mà trả về boolean. Ý tưởng đằng sau một ước lượng có chọn lọc liên kết là để đoán phần nào của các hàng trong một cặp bảng sẽ thỏa mãn một điều kiện mệnh đề WHERE ở dạng:
table1.column1 OP table2.column2

đối với toán tử hiện hành. Như với mệnh đề RESTRICT, điều này giúp trình tối ưu hóa rất đáng kể bằng việc cho phép nó chỉ ra cái nào trong vài tuần tự liên kết có thể có khả năng làm việc ít nhất.

Như trước đó, chương này sẽ không có ý định giải thích làm thế nào để viết một hàm ước lượng có lựa chọn liên kết, mà sẽ chỉ gợi ý rằng bạn sử dụng một trong các ước lượng tiêu chuẩn nếu có khả năng áp dụng được:

eqjoinsel cho =
neqjoinsel cho <>
scalarltjoinsel cho < hoặc <=
scalargtjoinsel cho > hoặc >=
areajoinsel cho các so sánh diện tích dựa vào 2D
positionjoinsel cho các so sánh vị trí dựa vào 2D
contjoinsel cho các so sánh phạm vi dựa vào 2D

35.13.5. HASHES

Mệnh đề HASHES (BĂM), nếu hiện diện, nói cho hệ thống rằng nó là cho phép để sử dụng phương pháp liên kết băm cho một liên kết dựa vào toán tử này. HASHES chỉ có ý nghĩa cho một toán tử nhị phân trả về boolean, và trong thực tế toán tử đó phải đại diện cho sự bằng như nhau đối với vài dạng dữ liệu hoặc cặp các dạng dữ liệu.

Giả thiết bên dưới sự liên kết băm là toán tử liên kết chỉ có thể trả về đúng (true) cho cặp các giá trị bên trái và bên phải băm đó cho mã băm y hệt. Nếu 2 giá trị đặt trong các chỗ băm khác nhau, thì liên kết đó sẽ không bao giờ so sánh chúng được, ngầm giả thiết là kết quả của toán tử liên kết đó phải là sai (false). Vì thế không bao giờ có ý nghĩa để chỉ định HASHES cho các toán tử không đại diện cho vài dạng ngang bằng nhau. Trong hầu hết các trường hợp chỉ là thực tế để hỗ trợ việc băm cho các toán tử mà lấy dạng dữ liệu y hệt ở cả 2 phía. Tuy nhiên, đôi lúc là có khả năng để thiết kế các hàm băm tương thích cho 2 hoặc nhiều dạng dữ liệu hơn; đó là, các hàm sẽ sinh ra các mã băm y hệt nhau cho các giá trị "bằng nhau", thậm chí dù các giá trị đó có các đại diện khác nhau. Ví dụ, là khá đơn giản để dàn xếp thuộc tính này khi băm các số nguyên của các độ rộng khác nhau.

Để được đánh dấu HASHES, toán tử liên kết phải xuất hiện trong một họ toán tử chỉ số băm. Điều này không bị ép buộc khi bạn tạo ra toán tử đó, vì tất nhiên việc tham chiếu họ toán tử còn chưa tồn tại. Nhưng các cố gắng sử dụng toán tử đó trong các liên kết băm sẽ thất bại trong thời gian chạy nếu không có họ toán tử nào như vậy tồn tại.

Hệ thống cần họ toán tử đó để tìm (các) hàm băm đặc thù dạng dữ liệu cho (các) dạng dữ liệu đầu vào của toán tử đó. Tất nhiên, bạn cũng phải tạo các hàm băm phù hợp trước khi bạn có thể tạo họ toán tử đó.

Sự chăm sóc nên được thực hiện khi chuẩn bị một hàm băm, vì sẽ có các cách thức phụ thuộc máy theo đó có thể thất bại để làm điều đúng. Ví dụ, nếu dạng dữ liệu của bạn là một cấu trúc theo đó có thể có các bit chêm vào không hay, thì bạn không thể đơn giản truyền toàn bộ cấu trúc đó tới hash_any. (Trừ phi bạn viết các toán tử và các hàm khác của bạn để đảm bảo rằng trong các máy mà đáp ứng được tiêu chuẩn dấu chấm động của IEEE, thì zero âm và zero dương là các giá trị khác nhau (các mẫu bit khác nhau) nhưng chúng được xác định để so sánh bằng nhau. Nếu một giá trị trôi nổi có thể chứa zero âm thì các bước bổ sung thêm là cần thiết để đảm bảo nó sinh ra giá trị băm y hệt như zero dương.

Một toán tử băm có khả năng liên kết phải có một hoán tử (bản thân nó nếu 2 dạng dữ liệu của toán hạng là y hệt nhau, hoặc một toán tử bằng có liên quan nếu chúng là khác nhau) xuất hiện trong cùng y hệt họ toán tử. Nếu điều này không phải thế, thì các lỗi của trình lên kế hoạch có thể xảy ra khi toán tử đó được sử dụng. Hơn nữa, là ý tưởng tốt (nhưng không nhất thiết được yêu cầu) cho một họ toán tử băm hỗ trợ nhiều dạng dữ liệu để cung cấp các toán tử ngang bằng nhau cho từng sự kết hợp các dạng dữ liệu; điều này cho phép tối ưu hóa tốt hơn.

> **Lưu ý**: Hàm nằm bên dưới một toán tử băm có khả năng liên kết phải được đánh dấu là không thể thay đổi được hoặc ổn định. Nếu nó là hay thay đổi, thì hệ thống sẽ không bao giờ có ý định sử dụng toán tử đó cho một liên kết băm.
>
> **Lưu ý**: Nếu một toán tử băm có khả năng liên kết được có một hàm nằm bên dưới mà được đánh dấu là khắt khe, thì hàm đó cũng phải là hoàn chỉnh: đó là, nó nên trả về đúng (true) hoặc sai (false), không bao giờ null, cho bất kỳ 2 đầu vào không null nào. Nếu qui tắc này không được tuân thủ, thì sự tối ưu hóa băm của các phép tính IN có thể sinh ra các kết quả sai. (Đặc biệt, IN có thể trả về sai nơi mà câu trả lời đúng theo tiêu chuẩn có thể là null; hoặc nó có thể lấy một lỗi kêu rằng nó không được chuẩn bị cho một kết quả null).

35.13.6. MERGES

Mệnh đề MERGES, nếu hiện diện, nói cho hệ thống rằng nó cho phép sử dụng phương pháp liên kết trộn cho một liên kết dựa vào toán tử này. MERGES chỉ có ý nghĩa cho một toán tử nhị phân mà trả về boolean, và trong thực tế toán tử đó phải đại diện cho sự ngăng bằng nhau đối với một vài dạng dữ liệu hoặc cặp các dạng dữ liệu.

Liên kết trộn dựa vào ý tưởng sắp xếp các bảng bên trái và bên phải vào trật tự và sau đó quét chúng song song. Vì thế, cả 2 dạng dữ liệu phải có khả năng xếp hàng được đầy đủ, và toán tử liên kết phải là toán tử chỉ có thể thành công đối với các cặp giá trị nằm trong "cùng một chỗ" trong trật

tự sắp xếp. Trong thực tế điều này có nghĩa là toán tử liên kết phải hành xử như sự ngang bằng nhau. Nhưng là có khả năng để liên kết trộn 2 dạng dữ liệu phân biệt nhau miễn là chúng tương thích nhau một cách logic. Ví dụ, toán tử bằng smallint vs. integer là có khả năng liên kết trộn được. Chúng ta chỉ cần sắp xếp các toán tử sẽ mang cả 2 dạng dữ liệu đó vào tuần tự tương thích về logic.

Để được đánh dấu là MERGES, thì toán tử liên kết phải xuất hiện như một thành viên ngang bằng của một họ toán tử có chỉ số btree. Điều này không bị ép buộc khi bạn tạo ra toán tử đó, vì tất nhiên việc tham chiếu họ toán tử có thể còn chưa tồn tại. Nhưng toán tử đó sẽ thực sự không được sử dụng cho các liên kết trộn trừ phi một họ toán tử khớp phù hợp có thể được tìm thấy. Cờ MERGES vì thế hành động như một mẹo để trình lên kế hoạch đáng tìm kiếm một họ toán tử khớp phù hợp.

Một toán tử có khả năng liên kết trộn phải có một hoán tử (bản thân nó nếu 2 dạng dữ liệu toán hạng là y hệt nhau, hoặc một toán tử ngang bằng có liên quan nếu chúng là khác nhau) xuất hiện trong cùng họ toán tử đó. Nếu điều này không phải thế, thì các lỗi của trình lên kế hoạch có thể xảy ra khi toán tử đó được sử dụng. Hơn nữa, là ý tưởng tốt (nhưng không được yêu cầu khắt khe) cho một họ toán tử btree mà hỗ trợ nhiều dạng dữ liệu để cung cấp các toán tử ngang bằng nhau cho từng sự kết hợp các dạng dữ liệu; điều này cho phép tối ưu hóa tốt hơn.

> **Lưu ý**: Hàm nằm bên dưới một toán tử có khả năng liên kết trộn phải được đánh dấu không thể thay đổi hoặc ổn định. Nếu nó là dễ thay đổi, thì hệ thống sẽ không bao giờ có ý định sử dụng toán tử đó cho một liên kết trộn.

35.14. Các mở rộng giao diện với các chỉ số

Các thủ tục được mô tả cho tới nay vì thế cho phép bạn định nghĩa các dạng mới, các hàm mới, và các toán tử mới. Tuy nhiên, chúng ta còn chưa thể định nghĩa một chỉ số trong một cột của một dạng dữ liệu mới. Để làm điều này, chúng ta phải định nghĩa một *lớp toán tử (operator class)* cho dạng dữ liệu mới đó. Sau đây trong phần này, chúng ta sẽ minh họa khái niệm này trong một ví dụ: một lớp toán tử mới cho phương pháp chỉ số B-tree lưu trữ và sắp xếp các số phức tạp theo trật tự giá trị tuyệt đối tăng dần.

Các lớp toán tử có thể được kết nhóm thành các *họ toán tử (operator family)* để chỉ ra các mối quan hệ giữa các lớp tương thích về ngữ nghĩa. Chỉ khi một dạng dữ liệu duy nhất có liên quan, một lớp toán tử là đủ, nên chúng ta tập trung vào trường hợp đó trước và sau đó trở về các họ toán tử.

35.14.1. Các phương pháp chỉ số và các lớp toán tử

Bảng pg_am chứa một hàng cho từng phương pháp chỉ số (nội bộ được biết như là phương pháp truy cập). Sự hỗ trợ truy cập thường xuyên tới các bảng được xây dựng trong PostgreSQL, nhưng tất cả các phương pháp chỉ số được mô tả trong pg_am. Có khả năng để thêm một phương pháp chỉ số mới bằng việc xác định các thủ tục giao diện được yêu cầu và sau đó tạo một hàng trong pg_am - nhưng điều đó nằm ngoài phạm vi của chương này (xem Phần 51).

Các thủ tục cho một phương pháp chỉ số không trực tiếp biết bất kỳ điều gì về các dạng dữ liệu mà phương pháp chỉ số đó sẽ vận hành. Thay vào đó, một *lớp toán tử* xác định tập hợp các tính toán mà

phương pháp chỉ số đó cần phải sử dụng để làm việc với một dạng dữ liệu đặc biệt. Các lớp toán tử được gọi thế vì một điều chúng chỉ định là tập hợp các toán tử mệnh đề WHERE mà có thể được sử dụng với một chỉ số (nghĩa là, có thể được chuyển đổi thành một sự định tính quét chỉ số). Một lớp toán tử cũng có thể chỉ định vài *thủ tục hỗ trợ* sẽ cần thiết đối với các tính toán nội bộ của phương pháp chỉ số đó, nhưng không trực tiếp ứng với bất kỳ toán tử mệnh đề WHERE nào mà có thể được sử dụng với chỉ số đó.

Là có khả năng để định nghĩa nhiều lớp toán tử cho phương pháp chỉ số và dạng dữ liệu y hệt. Bằng cách làm đó, nhiều tập hợp của việc đánh chỉ số ngữ nghĩa có thể được định nghĩa cho một dạng dữ liệu duy nhất. Ví dụ, một chỉ số B-tree đòi hỏi một trật tự sắp xếp phải được định nghĩa cho từng dạng dữ liệu mà nó làm việc trong đó. Có lẽ là hữu dụng cho một dạng dữ liệu số phức tạp để có một lớp toán tử B-tree mà sắp xếp các dữ liệu theo giá trị tuyệt đối phức tạp, dạng khác sắp xếp theo phần thực tế, và cứ như thế. Thông thường, một trong các lớp toán tử sẽ được cho rằng hữu dụng phổ biến nhất và sẽ được đánh dấu như là lớp toán tử mặc định cho dạng dữ liệu và phương pháp chỉ số đó.

Tên lớp toán tử y hệt đó có thể được sử dụng cho vài phương pháp chỉ số khác nhau (ví dụ, cả các phương pháp chỉ số B-tree và băm có các lớp toán tử có tên là `int4_ops`), nhưng từng lớp như vậy là một thực thể độc lập và phải được định nghĩa riêng rẽ.

35.14.2. Chiến lược phương pháp chỉ số

Các toán tử có liên quan tới một lớp toán tử được "các số chiến lược" xác định, phục vụ để nhận diện ngữ nghĩa của từng toán tử trong ngữ cảnh của lớp toán tử của nó. Ví dụ, B-tree áp đặt một trật tự khắt khe lên các khóa, từ nhỏ tới lớn, và vì thế các toán tử như "nhỏ hơn" và "lớn hơn hoặc bằng" là thú vị với lưu ý cho B-tree. Vì PostgreSQL cho phép người sử dụng định nghĩa các toán tử, PostgreSQL không thể xem tên của một toán tử (như, < hoặc >=) và nói đó là dạng so sánh nào. Thay vào đó, phương pháp chỉ số xác định một tập hợp "các chiến lược", có thể được nghĩ như các toán tử được tổng quát hóa. Từng lớp toán tử chỉ định toán tử thực sự nào tương ứng với từng chiến lược cho một dạng dữ liệu đặc biệt và sự giải thích các ngữ nghĩa chỉ số đó.

Phương pháp chỉ số B-tree định nghĩa 5 chiến lược, được chỉ ra trong Bảng 35-2.

Bảng 35-2. Các chiến lược B-tree

Phép tính	Số chiến lược
nhỏ hơn	1
nhỏ hơn hoặc bằng	2
bằng	3
lớn hơn hoặc bằng	4
lớn hơn	5

Các chỉ số băm chỉ hỗ trợ các so sánh bằng nhau, và vì thế chúng chỉ sử dụng chiến lược 1, được chỉ ra trong Bảng 35-3.

Bảng 35-3. Các chiến lược băm

Phép tính	Số chiến lược
bằng	1

Các chỉ số GiST là mềm dẻo hơn: chúng hoàn toàn không có một tập hợp cố định các chiến lược. Thay vào đó, thủ tục hỗ trợ "nhất quán" của từng lớp toán tử GiST đặc biệt giải nghĩa các số chiến lược mà nó giống. Như một ví dụ, vài lớp toán tử chỉ số GiST được xây dựng sẵn đánh chỉ số các đối tượng hình học 2 chiều, đưa ra các chiến lược "R-tree" được chỉ ra trong Bảng 35-4. Có 4 trong số đó là các kiểm thử 2 chiều đúng (gối lên nhau, y hệt, chứa, bị chứa); 4 trong số đó chỉ xem xét chiều X; và 4 thứ khác đưa ra các kiểm thử y hệt theo chiều Y.

Bảng 35-4 Các chiến lược "R-tree" 2 chiều của GiST

Phép tính	Số chiến lược
khắt khe trái của	1
không mở rộng sang phải của	2
gối lên nhau	3
không mở rộng sang trái của	4
khắt khe phải của	5
y hệt	6
chứa	7
được chứa bởi	8
không mở rộng trên	9
khắt khe dưới	10
khắt khe trên	11
không mở rộng dưới	12

Các chỉ số GIN là tương tự như các chỉ số GiST về độ mềm dẻo: chúng không có một tập hợp cố định các chiến lược. Thay vào đó các thủ tục hỗ trợ từng lớp toán tử giải nghĩa số chiến lược tuân theo định nghĩa lớp toán tử đó. Ví dụ, số chiến lược được các lớp được xây dựng sẵn sử dụng cho các mảng được chỉ ra trong Bảng 35-5.

Bảng 35-5. Các chiến lược mảng GIN

Phép tính	Số chiến lược
gối lên nhau	1
chứa	2
được chứa bởi	3
bằng	4

Lưu ý rằng tất cả các toán tử chiến lược trả về các giá trị Boolean. Trong thực tế, tất cả các toán tử được định nghĩa như là các chiến lược phương pháp chỉ số phải trả về dạng `boolean`, vì chúng phải xuất hiện ở mức đỉnh của mệnh đề WHERE sẽ được sử dụng với một chỉ số.

35.14.3. Các thủ tục hỗ trợ phương pháp chỉ số

Các chiến lược thường sẽ không đủ thông tin cho hệ thống để chỉ ra cách sử dụng một chỉ số. Trong thực tế, các phương pháp chỉ số đòi hỏi các thủ tục hỗ trợ bổ sung để làm việc. Ví dụ, phương pháp chỉ số B-tree phải có khả năng so sánh 2 khóa và xác định liệu một khóa có lớn hơn, hay bằng, hay nhỏ hơn khóa kia. Tương tự, phương pháp chỉ số băm phải có khả năng tính toán các mã băm cho các giá trị khóa. Các phép tính đó không tương ứng với các toán tử được sử dụng trong các định tính trong các lệnh SQL; chúng là các thủ tục quản trị được các phương pháp chỉ số sử dụng, một cách nội bộ.

Hệt như với các chiến lược, lớp toán tử nhận diện các hàm cụ thể nào sẽ đóng từng trong số các vai trò đó cho một dạng dữ liệu và giải nghĩa ngữ nghĩa được đưa ra. Phương pháp chỉ số đó định nghĩa tập hợp các hàm nó cần, và lớp toán tử xác định các hàm đúng để sử dụng bằng việc chỉ định chúng tới "các số hàm hỗ trợ" được phương pháp chỉ số đó chỉ định.

B-tree đòi hỏi một hàm hỗ trợ duy nhất, được chỉ ra trong Bảng 35-6.

Bảng 35-6. Các hàm hỗ trợ B-tree

Hàm	Số hỗ trợ
So sánh 2 khóa và trả về một số nguyên nhỏ hơn zero, zero hoặc lớn hơn zero, chỉ ra liệu khóa thứ nhất đó là nhỏ hơn, bằng, hoặc lớn hơn so với khóa thứ hai	1

Các chỉ số băm tương tự đòi hỏi một hàm hỗ trợ, được chỉ ra trong Bảng 35-7.

Bảng 35-7. Các hàm hỗ trợ băm

Hàm	Số hỗ trợ
Tính toán giá trị băm cho một khóa	1

Các chỉ số GiST đòi hỏi 7 hàm hỗ trợ, được chỉ ra trong Bảng 35-8.

Bảng 35-8. Các hàm hỗ trợ GiST

Hàm	Số hỗ trợ
nhất quán - xác định liệu khóa có thỏa mãn trình định tính truy vấn hay không	1
hợp nhất - tính sự hợp nhất của tập hợp các khóa	2
nén - tính đại diện được nén của một khóa hoặc giá trị sẽ được đánh chỉ số	3
giải nén - tính đại diện được giải nén của một khóa nén	4
hình phạt - tính hình phát cho việc chèn khóa mới vào cây con với khóa cây con được đưa ra	5
chọn chia (picksplit) - xác định các khoản nào của một trang sẽ phải bị chuyển tới trang mới và tính các khóa hợp nhất cho các trang kết quả	6
bằng - so sánh 2 khóa và trả về đúng (true) nếu chúng bằng nhau	7

Các chỉ số GIN đòi hỏi 4 hàm hỗ trợ, được chỉ ra trong Bảng 35-9.

Bảng 35-9. Các hàm hỗ trợ GIN

Hàm	Mô tả	Số hỗ trợ
compare	so sánh 2 khóa và trả về một số nguyên nhỏ hơn zero, zero, hoặc lớn hơn zero, chỉ ra liệu khóa thứ nhất có là nhỏ hơn, bằng, hoặc lớn hơn so với khóa thứ 2	1
extractValue	trích ra các khóa từ một giá trị sẽ được đánh chỉ số	2
extractQuery	trích ra các khóa từ một điều kiện truy vấn	3
consistent	xác định liệu giá trị có khớp với điều kiện truy vấn hay không	4
comparePartial	(phương pháp tùy chọn) so sánh khóa một phần từ truy vấn và khóa từ chỉ số, và trả về số nguyên nhỏ hơn zero, zero, hoặc lớn hơn zero, chỉ ra liệu GIN sẽ bỏ qua khoản chỉ số này hay không, đối xử với khoản đó như một sự trùng khớp, hoặc dừng quét chỉ số đó.	5

Không giống như các toán tử chiến lược, các hàm hỗ trợ trả về bất kỳ dạng dữ liệu nào mà phương pháp chỉ số đặc biệt kỳ vọng; ví dụ trong trường hợp hàm so sánh B-tree, một số nguyên được ký. Số và các dạng đối số cho từng hàm hỗ trợ cũng tương tự phụ thuộc vào phương pháp chỉ số. Đối với B-tree và băm thì các hàm hỗ trợ lấy các dạng dữ liệu đầu vào y hệt như các toán tử làm bao gồm trong lớp toán tử, nhưng điều này không phải thế cho hầu hết các hàm hỗ trợ GIN và GiST.

35.14.4. Ví dụ

Bây giờ chúng ta đã thấy các ý tưởng, ở đây là ví dụ có hứa hẹn về việc tạo ra một lớp toán tử. (Bạn có thể thấy một bản sao làm việc của ví dụ này trong src/tutorial/complex.c và src/tutorial/complex.sql trong phân phối nguồn). Lớp toán tử đó bọc complex_abs_ops. Trước nhất, chúng ta cần một tập hợp các toán tử. Thủ tục cho việc xác định các toán tử đã được thảo luận trong Phần 35.12. Đối với một lớp toán tử trong B-tree, các toán tử chúng ta yêu cầu là:

- giá trị tuyệt đối nhỏ hơn (chiến lược 1)
- giá trị tuyệt đối nhỏ hơn hoặc bằng (chiến lược 2)
- giá trị tuyệt đối bằng (chiến lược 3)
- giá trị tuyệt đối lớn hơn hoặc bằng (chiến lược 4)
- giá trị tuyệt đối lớn hơn (chiến lược 5)

Cách có xu hướng lỗi ít nhất để định nghĩa một tập hợp các toán tử so sánh có liên quan là viết hàm hỗ trợ so sánh B-tree trước, và sau đó viết các hàm khác như các trình gói 1 dòng xung quanh hàm hỗ trợ đó. Điều này làm giảm những xung đột có các kết quả không nhất quán cho các trường hợp góc. Theo tiếp cận này, chúng ta trước nhất viết:

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
        double amag = Mag(a),
        bmag = Mag(b);
```

```
        if (amag < bmag)
                return -1;
        if (amag > bmag)
                return 1;
        return 0;
}
```

Bây giờ hàm nhỏ hơn trông giống thế này:

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
        Complex *a = (Complex *) PG_GETARG_POINTER(0);
        Complex *b = (Complex *) PG_GETARG_POINTER(1);

        PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

4 hàm khác chỉ trong cách mà chúng so sánh kết quả hàm nội bộ với zero.

Tiếp sau chúng ta khai báo các hàm và các toán tử dựa vào các hàm cho SQL:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
        AS 'filename', 'complex_abs_lt'
        LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
        leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
        commutator = > , negator = >= ,
        restrict = scalarltsel, join = scalarltjoinsel
);
```

Điều quan trọng phải chỉ định hoán tử và các toán tử phủ định đúng, cũng như các hàm hạn chế và chọn lọc liên kết phù hợp, nếu không thì trình tối ưu hóa sẽ không có khả năng sử dụng có hiệu quả chỉ số đó. Lưu ý là các trường hợp nhỏ hơn, bằng và lớn hơn sử dụng các hàm chọn lựa khác nhau.

Những điều khác đáng lưu ý đang xảy ra ở đây:

- Chỉ có một toán tử được đặt tên, như, = và lấy dạng `complex` cho cả 2 toán hạng. Trong trường hợp này chúng ta không có bất kỳ toán tử nào khác = đối với `complex`, nhưng nếu chúng ta đã từng xây dựng một dạng dữ liệu thực tế thì chúng ta có thể muốn = sẽ là sự tính toán bằng thông thường cho các số phức tạp (và không bằng đối với các giá trị tuyệt đối). Trong trường hợp đó, chúng ta có lẽ cần sử dụng vài tên toán tử khác cho `complex_abs_eq`.
- Dù PostgreSQL có thể đối phó được với các hàm có tên SQL cùng y hệt miễn là chúng có các dạng dữ liệu đối số khác nhau, thì C chỉ có thể đối phó được với một hàm tổng thể có một tên được đưa ra. Vì thế chúng ta sẽ không đặt tên cho hàm C thứ gì đó đơn giản như `abs_eq`. Thường thì là thực tiễn tốt để đưa vào tên dạng dữ liệu trong tên hàm C, sao cho không xung đột với các hàm cho các dạng dữ liệu khác.
- Chúng ta có thể đã tạo ra tên SQL của hàm `abs_eq`, dựa vào PostgreSQL để phân biệt nó bằng các dạng dữ liệu đối số từ bất kỳ hàm SQL nào khác của tên y hệt. Để giữ cho ví dụ

đơn giản, chúng ta tạo hàm có các tên y hệt ở mức C và mức SQL.

Bước tiếp theo là đăng ký thủ tục hỗ trợ được B-tree yêu cầu. Ví dụ mã C triển khai điều này là trong cùng tệp y hệt mà có chứa các hàm toán tử. Đây là cách chúng ta khai báo hàm đó:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

Bây giờ chúng ta có các toán tử và các thủ tục hỗ trợ được yêu cầu, chúng ta cuối cùng có thể tạo lớp toán tử:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR 1 < ,
        OPERATOR 2 <= ,
        OPERATOR 3 = ,
        OPERATOR 4 >= ,
        OPERATOR 5 > ,
        FUNCTION 1 complex_abs_cmp(complex, complex);
```

Và chúng ta đã thực hiện xong! Bây giờ có khả năng tạo và sử dụng các chỉ số B-tree trong các cột complex.

Chúng ta có thể đã viết các khoản đầu vào toán tử dài dòng hơn, như trong:

```
OPERATOR 1 < (complex, complex),
```

nhưng không có nhu cầu phải làm thế khi các toán tử lấy dạng dữ liệu y hệt chúng ta đang định nghĩa lớp toán tử đó.

Ví dụ ở trên giả thiết là bạn muốn biến lớp toán tử mới này thành lớp toán tử B-tree mặc định cho dạng dữ liệu complex. Nếu bạn không làm, hãy để ra ngoài từ mặc định DEFAULT.

35.14.5. Lớp toán tử và họ toán tử

Cho tới nay chúng ta đã ngầm giả thiết rằng một lớp toán tử làm việc với chỉ một dạng dữ liệu. Trong khi nhất định có thể chỉ là một dạng dữ liệu trong một cột chỉ số đặc biệt, thì thường là hữu dụng để đánh chỉ số các tính toán so sánh một cột được đánh chỉ số với một giá trị của một dạng dữ liệu khác. Hơn nữa, nếu có sử dụng cho một toán tử dạng liên dữ liệu trong sự kết nối với một lớp toán tử, thì thường là trường hợp dạng dữ liệu khác đó có một lớp toán tử có liên quan của riêng nó. Là hữu dụng để làm cho các kết nối giữa các lớp có liên quan rõ ràng, vì điều này có thể giúp trình lên kế hoạch trong việc tối ưu các truy vấn SQL (đặc biệt cho các lớp toán tử B-tree, vì trình lên kế hoạch có nhiều tri thức về cách để làm việc với chúng).

Để điều khiển các nhu cầu đó, PostgreSQL sử dụng khái niệm *họ toán tử*. Một họ toán tử có một hoặc nhiều lớp toán tử, và cũng có thể có các toán tử có khả năng đánh chỉ số được và các hàm hỗ trợ tương ứng thuộc về họ đó như một tổng thể nhưng không thuộc về bất kỳ lớp duy nhất nào trong họ đó. Chúng ta nói rằng các toán tử và các hàm như vậy là "lỏng lẻo" với họ đó, như là ngược với bị ràng buộc trong một lớp nhất định nào đó. Thường thì mỗi lớp toán tử có chứa các toán tử dạng dữ liệu đơn nhất, trong khi các toán tử dạng liên dữ liệu là lỏng lẻo trong họ đó.

Tất cả các toán tử và các hàm trong một họ toán tử phải có các ngữ nghĩa tương thích nhau, trong đó các yêu cầu về tính tương thích được phương pháp đánh chỉ số thiết lập. Bạn có lẽ vì thế ngạc nhiên vì sao lo lắng tách riêng ra các tập con đặt biệt của họ đó như là các lớp toán tử; và quả thực vì nhiều lý do cho việc xác định các lớp toán tử sao cho chúng chỉ định họ đó là cần thiết bao nhiêu để hỗ trợ cho bất kỳ chỉ số đặc biệt nào. Nếu có một chỉ số sử dụng một lớp toán tử, thì lớp toán tử đó không thể bị bỏ đi mà không có việc bỏ đi chỉ số đó - nhưng các phần khác của họ toán tử đó, ấy là các lớp toán tử và các toán tử lỏng lẻo khác, có thể bị bỏ đi. Vì thế, một lớp toán tử nên được chỉ định để bao gồm tập hợp tối thiểu các toán tử và các hàm là cần thiết hợp lý để làm việc với một chỉ số trong một dạng dữ liệu đặc thù, và sau đó các toán tử có liên quan nhưng không cơ bản có thể được thêm vào như là các thành viên lỏng lẻo của họ toán tử đó.

Như một ví dụ, PostgreSQL có một họ toán tử B-tree được xây sẵn integer_ops, nó bao gồm các lớp toán tử int8_ops, int4_ops, và int2_ops cho các chỉ số trong các cột bigint (int8), integer (int4), và smallint (int2) một cách tương ứng. Họ đó cũng chứa các toán tử so sánh dạng liên dữ liệu cho phép 2 dạng dữ liệu bất kỳ đó sẽ được so sánh, sao cho một chỉ số trong 1 trong số các dạng đó có thể tìm kiếm được bằng việc sử dụng một giá trị so sánh của dạng khác. Họ đó có thể được đúp bản bằng các định nghĩa đó:

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
        DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
        -- standard int8 comparisons
        OPERATOR 1 < ,
        OPERATOR 2 <= ,
        OPERATOR 3 = ,
        OPERATOR 4 >= ,
        OPERATOR 5 > ,
        FUNCTION 1 btint8cmp(int8, int8) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
        -- standard int4 comparisons (các so sánh int4 tiêu chuẩn)
        OPERATOR 1 < ,
        OPERATOR 2 <= ,
        OPERATOR 3 = ,
        OPERATOR 4 >= ,
        OPERATOR 5 > ,
        FUNCTION 1 btint4cmp(int4, int4) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
        -- standard int2 comparisons
        OPERATOR 1 < ,
        OPERATOR 2 <= ,
        OPERATOR 3 = ,
        OPERATOR 4 >= ,
        OPERATOR 5 > ,
        FUNCTION 1 btint2cmp(int2, int2) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
        -- cross-type comparisons int8 vs int2 (các so sánh dạng liên int8 vs int2)
        OPERATOR 1 < (int8, int2) ,
        OPERATOR 2 <= (int8, int2) ,
        OPERATOR 3 = (int8, int2) ,
```

```
OPERATOR 4 >= (int8, int2) ,
OPERATOR 5 > (int8, int2) ,
FUNCTION 1 btint82cmp(int8, int2) ,

-- cross-type comparisons int8 vs int4 (các so sánh dạng liên int8 vs int4)
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- cross-type comparisons int4 vs int2 (các so sánh dạng liên int4 vs int2)
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- cross-type comparisons int4 vs int8 (các so sánh dạng liên int4 vs int8)
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- cross-type comparisons int2 vs int8 (các so sánh dạng liên int2 vs int8)
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,


-- cross-type comparisons int2 vs int4 (các so sánh dạng liên int2 vs int4)
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

Lưu ý là định nghĩa này "quá tải" chiến lược toán tử và các số hàm hỗ trợ: từng số xảy ra nhiều lần trong họ đó. Điều này được phép miễn là từng trường hợp của một số đặc biệt có các dạng dữ liệu đầu vào phân biệt được. Các trường hợp có cả các dạng đầu vào bằng nhau cho một dạng đầu vào lớp toán tử là các toán tử và các hàm hỗ trợ ban đầu cho lớp toán tử đó, và trong hầu hết các trường hợp sẽ được khai báo như một phần của lớp toán tử thay vì như các thành viên lỏng lẻo của họ đó.

Trong một họ toán tử B-tree, tất cả các toán tử trong họ đó phải phân loại tương thích, nghĩa là các luật bắc cầu xuyên suốt tất cả các dạng dữ liệu được họ đó hỗ trợ: "nếu A = B và B = C, thì A = C", và "nếu A < B và B < C, thì A < C". Đối với từng toán tử trong họ đó phải có một hàm hỗ trợ có 2 dạng dữ liệu đầu vào y hệt như toán tử đó. Được khuyến cáo rằng một họ là hoàn chỉnh, nghĩa là, đối với từng sự kết hợp các dạng dữ liệu, tất cả các toán tử được bao gồm. Từng lớp toán tử sẽ bao

gồm chỉ các toán tử và hàm hỗ trợ không phải liên dạng cho dạng dữ liệu của nó.

Để xây dựng một họ toán tử băm nhiều dạng dữ liệu (multiple-data-type), các hàm hỗ trợ băm tương thích phải được tạo ra cho từng dạng dữ liệu được họ đó hỗ trợ. Tính tương thích ở đây nghĩa là các hàm được đảm bảo trả về mã băm y hệt cho bất kỳ 2 giá trị nào mà được coi là bằng nhau đối với các toán tử bằng nhau của họ đó, thậm chí khi các giá trị đó có các dạng khác nhau. Đây là khó khăn thường thấy để hoàn tất khi các dạng có các đại diện vật lý khác nhau, nhưng điều đó có thể được làm trong vài trường hợp. Lưu ý rằng chỉ có một hàm hỗ trợ cho mỗi dạng dữ liệu, không phải một hàm cho từng toán tử bằng nhau. Được khuyến cáo rằng một họ sẽ là hoàn chỉnh, nghĩa là, đưa ra một toán tử bằng nhau cho từng kết hợp các dạng dữ liệu. Từng lớp toán tử nên bao gồm chỉ toán tử và hàm hỗ trợ bằng nhau không phải dạng liên dữ liệu cho dạng dữ liệu của mình.

Các chỉ số GIN và GiST không có bất kỳ ký hiệu rõ ràng nào đối với các tính toán dạng liên dữ liệu. Tập hợp các toán tử được hỗ trợ chỉ là bất kỳ điều gì các hàm hỗ trợ ban đầu cho một lớp toán tử được đưa ra có thể điều khiển.

> **Lưu ý**: Trước phiên bản PostgreSQL 8.3, đã không có khái niệm về các họ toán tử, và vì thế bất kỳ các toán tử dạng liên dữ liệu nào có ý định được sử dụng với một chỉ số đã phải ràng buộc trực tiếp trong lớp toán tử của một chỉ số. Trong khi tiếp cận này vẫn còn làm việc, thì nó bị phản đối vì nó làm cho các phụ thuộc của một chỉ số là quá rộng, và vì trình lên kế hoạch có thể điều khiển các so sánh dạng liên dữ liệu có hiện quả hơn khi cả 2 dạng dữ liệu có các toán tử trong cùng họ toán tử.

35.14.6. Các phụ thuộc hệ thống trong các lớp toán tử

PostgreSQL sử dụng các lớp toán tử để suy luận các thuộc tính của các toán tử theo nhiều cách thức hơn so với chỉ liệu chúng có thể được sử dụng với các chỉ số hay không. Vì thế, bạn có lẽ muốn tạo ra các lớp toán tử thậm chí nếu bạn không định đánh chỉ số bất kỳ cột nào dạng có dữ liệu của bạn.

Đặc biệt, có các tính năng SQL như ORDER BY và DISTINCT đòi hỏi sự so sánh và sắp xếp các dữ liệu. Để triển khai các tính năng đó trong một dạng dữ liệu do người sử dụng định nghĩa, PostgreSQL tìm kiếm lớp toán tử B-tree mặc định cho dạng dữ liệu đó. Thành viên "bằng nhau" của lớp toán tử đó xác định dấu bằng các giá trị của hệ thống cho GROUP BY và DISTINCT, và trật tự sắp xếp được lớp toán tử đó đặt ra xác định trật tự mặc định ORDER BY.

So sánh các mảng các dạng do người sử dụng định nghĩa cũng dựa vào ngữ nghĩa được lớp toán tử B-tree mặc định xác định.

Nếu không có lớp toán tử B-tree mặc định cho một dạng dữ liệu, thì hệ thống sẽ tìm kiếm một lớp toán tử băm mặc định. Nhưng vì dạng lớp toán tử đó chỉ cung cấp sự bằng nhau, nên trong thực tế chỉ là đủ để hỗ trợ tính bằng nhau của mảng.

Khi không có lớp toán tử mặc định cho một dạng dữ liệu, thì bạn sẽ có các lỗi như "không thể nhận diện một toán tử sắp xếp trật tự" (could not identify an ordering operator) nếu bạn cố sử dụng các tính năng SQL đó với dạng dữ liệu đó.

> **Lưu ý**: Trong các phiên bản PostgreSQL trước 7.4, việc sắp xếp và lập nhóm các tính toán có thể ngầm sử dụng các toán tử có tên =, <, và >. Hành vi mới của việc dựa vào các lớp

toán tử mặc định tránh tạo ra bất kỳ giả thiết nào về hành vi của các toán tử với tên đặc biệt.

Một điểm quan trọng khác là một toán tử xuất hiện trong một họ toán tử băm là một ứng viên cho các liên kết băm, tổng hợp băm, và tối ưu hóa có liên quan. Họ toán tử băm là cơ bản ở đây vì nó nhận diện (các) hàm băm để sử dụng.

35.14.7. Tính năng đặc biệt của các lớp toán tử

Có 2 tính năng đặc biệt của các lớp toán tử mà chúng ta còn chưa thảo luận, chủ yếu vì chúng không hữu dụng với các phương pháp chỉ số được sử dụng phổ biến nhất.

Thông thường, việc khai báo một toán tử như một thành viên của một lớp toán tử (hoặc họ) có nghĩa là phương pháp chỉ số đó có thể truy xuất chính xác tập hợp các hàng thỏa mãn một điều kiện WHERE bằng việc sử dụng toán tử đó. Ví dụ:

```
SELECT * FROM table WHERE integer_column < 4;
```

có thể chính xác được thỏa mãn bằng một chỉ số B-tree trong cột số nguyên. Nhưng có những trường hợp nơi mà một chỉ số là hữu dụng như một chỉ dẫn không chính xác cho việc khớp các hàng. Ví dụ, nếu một chỉ số GiST lưu trữ chỉ các hộp ràng buộc cho các đối tượng hình học, thì nó không thể chính xác làm thỏa mãn một điều kiện WHERE mà kiểm thử sự gối lên nhau giữa các đối tượng không phải hình tam giác như các đa giác. Vâng chúng ta có thể sử dụng chỉ số đó để tìm các đối tượng mà hộp ràng buộc của nó gối lên hộp ràng buộc của đối tượng đích, và sau đó làm chính xác kiểm thử gối lên chỉ trong các đối tượng được chỉ số đó tìm thấy. Nếu kịch bản này áp dụng được, thì chỉ số đó được nói sẽ là "thiệt hại" đối với toán tử đó. Các tìm kiếm chỉ số thiệt hại được triển khai bằng việc có phương pháp chỉ số trả về một cờ *tái kiểm tra* khi một hàng có thể có hoặc không thực sự làm thỏa mãn điều kiện của truy vấn. Hệ thống điểm sau đó sẽ kiểm thử điều kiện truy vấn ban đầu trong hàng được truy xuất để xem liệu nó sẽ có trả về như một sự trùng khớp hợp lệ hay không. Tiếp cận này làm việc nếu chỉ số đó được đảm bảo trả về tất cả các hàng được yêu cầu, cộng với có thể vài hàng bổ sung thêm, có thể bị loại trừ bằng việc thực thi triệu gọi toán tử ban đầu. Các phương pháp chỉ số hỗ trợ cho các tìm kiếm thiệt hại (hiện hành, GiST và GIN) cho phép các hàm hỗ trợ các lớp toán tử riêng rẽ thiết lập cờ tái kiểm tra, và vì thế điều này cơ bản là một tính năng của lớp toán tử.

Cân nhắc một lần nữa tình huống nơi mà chúng ta đang lưu trữ trong chỉ số đó chỉ hộp ràng buộc của một đối tượng phức tạp như một đa giác. Trong trường hợp này không có nhiều giá trị trong việc lưu trữ toàn bộ đa giác đó trong khoản chỉ số - chúng ta có lẽ cũng lưu chỉ một đối tượng đơn giản hơn dạng hộp box. Tình huống này được lựa chọn STORAGE thể hiện trong CREATE OPERATOR CLASS: chúng ta muốn viết thứ gì đó giống như:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

Hiện tại, chỉ các phương pháp chỉ số GiST và GIN hỗ trợ một dạng STORAGE mà là khác với dạng dữ liệu cột. Các thủ tục hỗ trợ compress và decompress của GiST phải làm việc với sự chuyển đổi

dạng dữ liệu khi STORAGE được sử dụng. Trong GIN, dạng STORAGE nhận diện dạng các giá trị "khóa", thường là khác với dạng cột được đánh chỉ số - ví dụ, một lớp toán tử cho các cột mảng số nguyên có thể có các khóa sẽ chỉ là các số nguyên. Các thủ tục hỗ trợ extractValue và extractQuery của GIN có trách nhiệm cho việc trích xuất các khóa từ các giá trị được đánh chỉ số.

35.15. Sử dụng C++ cho khả năng mở rộng

Có khả năng sử dụng một trình biên dịch trong chế độ C++ để xây dựng các mở rộng PostgreSQL bằng việc tuân theo các chỉ dẫn sau:

- Tất cả các hàm được phần phụ trợ (backend) truy cập phải trình bày một giao diện C tới phần phụ trợ đó; Các hàm C có thể sau đó gọi các hàm C++. Ví dụ, kết nối extern C được yêu cầu cho các hàm được phần phụ trợ truy cập. Điều này cũng là cần thiết cho bất kỳ hàm nào được truyền như các con trỏ giữa phần phụ trợ và mã của C++.
- Bộ nhớ tự do sử dụng phương pháp bỏ phân bổ phù hợp. Ví dụ, hầu hết bộ nhớ phần phụ trợ được phân bổ bằng việc sử dụng palloc(), nên hãy sử dụng pfree() để giải phóng nó, nghĩa là việc sử dụng C++ delete() trong các trường hợp như vậy sẽ thất bại.
- Hãy ngăn ngừa các loại trừ từ việc truyền giống vào mã C (sử dụng một khối chộp tất cả ở mức đỉnh của tất cả các hàm extern C). Điều này là cần thiết thậm chí nếu mã C++ không đưa ra bất kỳ sự loại trừ nào vì các sự kiện giống như tràn bộ nhớ vẫn đưa ra các loại trừ. Bất kỳ sự loại trừ nào cũng phải nắm bắt được và các lỗi phù hợp được truyền ngược lại tới giao diện C. Nếu có thể, hãy biên dịch C++ với -fno-exceptions để loại bỏ hoàn toàn các loại trừ; trong các trường hợp như vậy, bạn phải kiểm tra các thất bại trong mã C++, như hãy kiểm tra NULL được new() trả về.
- Nếu gọi các hàm phụ trợ từ mã C++, hãy chắc chắn rằng kho các cuộc gọi C++ có không chỉ các cấu trúc dữ liệu thô cũ - POD (Plain Old Data). Điều này là cần thiết vì các lỗi phần phụ trợ sinh ra một longjmp() xa cách không mở đúng một kho cuộc gọi C++ với các đối tượng không phải POD.

Tóm lại, là tốt nhất để đặt mã C bên dưới một bức tường của các hàm extern C mà giao diện với phần phụ trợ, và tránh loại trừ, bộ nhớ, và rò rỉ kho cuộc gọi.

Chương 36. Trigger

Chương này đưa ra thông tin chung về việc viết các hàm trigger. Các hàm trigger có thể được viết trong hầu hết các ngôn ngữ thủ tục có sẵn, bao gồm cả PL/pgSQL (Chương 39), PL/Tcl (Chương 40), PL/Perl (Chương 41), và PL/Python (Chương 42). Sau khi đọc chương này, bạn nên tư vấn chương cho ngôn ngữ thủ tục ưa thích của bạn để tìm ra các chi tiết đặc thù ngôn ngữ đó để viết một trigger trong nó.

Cũng có khả năng để viết một hàm trigger trong C, dù hầu hết mọi người thấy là dễ dàng hơn để sử dụng một trong các ngôn ngữ thủ tục. Hiện không có khả năng để viết một hàm trigger trong ngôn ngữ hàm SQL thông thường.

36.1. Tổng quan về hành vi của trigger

Một trigger là một đặc tả mà cơ sở dữ liệu nên tự động thực thi một hàm đặc biệt bất kỳ khi nào một dạng tính toán nhất định được thực hiện. Các trigger có thể được xác định để thực thi hoặc trước hoặc sau bất kỳ hoạt động INSERT, UPDATE, hoặc DELETE nào, hoặc một lần cho từng hàng được sửa đổi, hoặc một lần cho một lệnh SQL. Các trigger UPDATE có thể được thiết lập để chạy chỉ nếu các cột nhất định được nhắc trong mệnh đề SET của lệnh UPDATE. Các trigger cũng có thể chạy cho các lệnh TRUNCATE. Nếu một sự kiện trigger xảy ra, thì hàm trigger được gọi ở thời điểm phù hợp để điều khiển sự kiện đó.

Hàm trigger phải được định nghĩa trước khi bản thân trigger đó có thể được tạo ra. (Hàm trigger nhận đầu vào của nó qua một cấu trúc TriggerData được truyền đặc biệt, không ở dạng của các đối số hàm thông thường).

Một khi một hàm trigger phù hợp đã được tạo ra, thì trigger đó được thiết lập với CREATE TRIGGER. Hàm trigger y hệt có thể được sử dụng cho nhiều trigger.

PostgreSQL đưa ra cả các trigger *theo từng hàng* và các trigger *theo từng lệnh*. Với một trigger theo từng hàng, thì hàm trigger đó được gọi một lần cho từng hàng bị ảnh hưởng vì lệnh làm chạy trigger đó. Ngược lại, một trigger theo từng lệnh được gọi một lần chỉ khi một lệnh phù hợp được thực thi, bất kể số hàng bị lệnh đó gây ảnh hưởng. Đặc biệt, một lệnh ảnh hưởng tới zero hàng sẽ vẫn gây ra sự thực thi của bất kỳ trigger theo từng lệnh áp dụng được nào. Hai dạng các trigger đó đôi khi được gọi là các trigger *mức hàng* và các trigger *mức lệnh*, một cách tương ứng. Các trigger trong TRUNCATE cũng có thể được xác định ở mức lệnh.

Các trigger cũng được phân loại là các trigger *trước* và trigger *sau*. Các trigger trước mức lệnh về tự nhiên chạy trước khi lệnh đó bắt đầu làm bất kỳ điều gì, trong khi các trigger sau mức lệnh chạy ở tận cuối của lệnh. Các trigger trước mức hàng chạy ngay lập tức trước khi một hàng cụ thể được vận hành, trong khi các trigger sau mức hàng chạy ở cuối lệnh (nhưng trước khi bất kỳ trigger sau mức lệnh nào).

Các hàm trigger được các trigger theo từng lệnh triệu gọi luôn trả về NULL. Các hàm trigger được các trigger triệu gọi có thể trả về một hàng của bảng (một giá trị dạng HeapTuple) cho trình thực thi

lời gọi đó, nếu chúng chọn. Một trigger mức hàng chạy trước một hoạt động có các lựa chọn sau:

- Nó có thể trả về NULL để bỏ qua hoạt động đối với hàng hiện hành. Điều này ra lệnh cho trình thực thi không thực hiện hoạt động mức hàng mà đã triệu gọi trigger đó (sự chèn hoặc sửa đổi một hàng của bản đặc biệt).
- Chỉ cho các trigger INSERT và UPDATE mức hàng, hàng được trả về *sẽ* trở thành hàng mà sẽ được chèn hoặc sẽ thay thế hàng đang được cập nhật. Điều này cho phép hàm trigger sửa đổi hàng đang được chèn hoặc cập nhật.

Một trigger trước mức hàng không có ý định gây ra hoặc các hành vi đó phải thận trọng để trả về như là kết quả của nó hàng y hệt đã từng được truyền trong đó (đó là, hàng NEW cho các trigger INSERT và UPDATE, hàng OLD cho các trigger DELETE).

Giá trị trả về bị bỏ qua cho các trigger mức hàng được chạy sau một hoạt động, và vì thế chúng có thể trả về NULL.

Nếu nhiều hơn một trigger được định nghĩa cho cùng y hệt một sự kiện trong mối quan hệ y hệt, thì các trigger sẽ được chạy theo trật tự abc theo tên trigger. Trong trường hợp các trigger trước, thì hàng có khả năng được sửa đổi được trả về bằng từng trigger sẽ trở thành đầu vào cho trigger tiếp sau. Nếu bất kỳ trigger trước nào trả về NULL, thì hoạt động đó bị bỏ đối với hàng đó và các trigger tiếp sau sẽ không được chạy.

Một định nghĩa trigger cũng có thể chỉ định một điều kiện Boolean WHEN, nó sẽ được kiểm thử để xem liệu trigger đó có được chạy hay không. Trong các trigger mức hàng thì điều kiện WHEN có thể kiểm tra các giá trị cũ và/hoặc mới của các cột của hàng đó. (Các trigger mức lệnh cũng có thể có điều kiện WHEN, dù tính năng đó không thật hữu dụng đối với chúng). Trong một trigger trước, điều kiện WHEN được đánh giá ngay trước khi hàm đó được hoặc có thể được chạy, vì thế việc sử dụng WHEN là không khác về mặt vật chất với việc kiểm thử điều kiện y hệt ở đầu của hàm trigger đó. Tuy nhiên, trong một trigger sau, thì điều kiện WHEN được đánh giá ngay sau khi cập nhật hàng đó xảy ra, và nó xác định liệu một sự kiện có nằm trong hàng đợi để chạy trigger đó ở cuối lệnh hay không. Vì thế khi một điều kiện WHEN của một trigger sau không trả về đúng (true), thì không cần thiết để cho vào hàng đợi một sự kiện cũng không lấy lại được hàng đó ở cuối của lệnh. Điều này có thể gây ra sự tăng tốc đáng kể trong các lệnh sửa đổi nhiều hàng, nếu trigger đó chỉ cần được chạy cho một ít hàng.

Thông thường, các trigger trước hàng được sử dụng để kiểm tra hoặc sửa đổi dữ liệu mà sẽ được chèn hoặc cập nhật. Ví dụ, một trigger trước có thể được sử dụng để chèn thời gian hiện hành vào một cột timestamp, hoặc kiểm tra xem 2 phần tử của hàng có nhất quán hay không. Các trigger sau hàng được sử dụng hợp lý nhất để nhân giống các cập nhật tới các bảng khác, hoặc thực hiện kiểm tra độ ổn định đối với các bảng khác. Lý do cho sự phân chia công việc này là một trigger sau có thể chắc chắn nó đang thấy giá trị cuối cùng của hàng đó, trong khi một trigger trước không thể; có thể có các trigger trước khác chạy sau nó. Nếu bạn không có lý do đặc biệt nào để làm một trigger trước hoặc sau, thì trường hợp sau là hiệu quả hơn, vì thông tin về hoạt động không phải lưu lại cho tới cuối lệnh.

Nếu một hàm trigger thực thi các lệnh SQL thì sau đó các lệnh đó có thể chạy các trigger một lần nữa. Điều này được biết tới như là các trigger nối tầng. Không có giới hạn trực tiếp nào về số các mức nối tầng. Có khả năng đối với các nối tầng để gây ra một lời gọi đệ qui của chính trigger đó; ví dụ, một trigger INSERT có thể thực thi một lệnh chèn một hàng bổ sung thêm vào bảng y hệt, làm cho trigger INSERT đó sẽ được chạy một lần nữa. Là trách nhiệm của lập trình viên trigger đó để tránh sự đệ qui bất tận trong các kịch bản như vậy.

Khi một trigger đang được định nghĩa, các đối số có thể được chỉ định cho nó. Mục đích của việc đưa vào các đối số trong định nghĩa trigger là để cho phép các trigger khác nhau với các yêu cầu tương tự nhau để gọi hàm y hệt đó. Ví dụ, có thể có một hàm trigger được tổng quát hóa mà lấy như là các đối số 2 tên cột và đặt người sử dụng hiện hành vào một tên và dấu thời gian hiện hành vào tên kia. Được viết đúng, hàm trigger này có thể là độc lập đối với bảng đặc thù mà nó đang chạy trigger trong đó. Vì thế hàm y hệt có thể được sử dụng cho các sự kiện INSERT trong bất kỳ bảng nào với các cột phù hợp, để tự động lần vết sự tạo các bản ghi trong một bảng giao dịch, ví dụ thế. Nó cũng có thể được sử dụng để theo dõi các sự kiện cập nhật mới nhất nếu được định nghĩa như là một trigger UPDATE.

Từng ngôn ngữ lập trình hỗ trợ các trigger có phương pháp riêng của mình cho việc làm cho dữ liệu đầu vào của trigger sẵn sàng cho hàm trigger đó. Dữ liệu đầu vào này bao gồm dạng sự kiện trigger (như, INSERT hoặc UPDATE) cũng như bất kỳ đối số nào mà đã được liệt kê trong CREATE TRIGGER. Đối với trigger mức hàng, dữ liệu đầu vào cũng bao gồm hàng NEW cho các trigger INSERT và UPDATE, và/hoặc hàng OLD cho các trigger UPDATE và DELETE. Các trigger mức lệnh hiện không có bất kỳ cách gì để kiểm tra (các) hàng riêng rẽ được lệnh đó sửa đổi.

36.2. Khả năng nhìn thấy các thay đổi dữ liệu

Nếu bạn thực thi các lệnh SQL trong hàm trigger của bạn, và các lệnh đó truy cập bảng mà trigger đó tác động tới, thì bạn cần nhận thức được về các qui tắc về khả năng thấy dữ liệu, vì chúng xác định liệu các lệnh SQL đó sẽ có nhìn thấy sự thay đổi dữ liệu mà trigger đó được chạy hay không. Ngắn gọn:

- Các trigger mức lệnh tuân theo các qui tắc về khả năng nhìn thấy đơn giản: không thay đổi nào được thực hiện bằng một lệnh là nhìn thấy được đối với các trigger mức lệnh sẽ được gọi trước lệnh đó, trong khi tất cả các sửa đổi là nhìn thấy đối với các trigger sau mức lệnh.
- Sự thay đổi dữ liệu (chèn, cập nhật, hoặc xóa) làm cho trigger đó chạy, một cách tự nhiên là không nhìn thấy được đối với các lệnh SQL được thực thi trong trigger trước mức hàng, vì nó còn chưa xảy ra.
- Tuy nhiên, các lệnh SQL được thực hiện trong một trigger trước mức hàng sẽ thấy các hiệu ứng thay đổi dữ liệu cho các hàng trước khi được xử lý trong lệnh bên ngoài y hệt. Điều này đòi hỏi sự thận trọng, vì trật tự của các sự kiện thay đổi đó không nằm trong sự đoán định chung trước được; một lệnh SQL ảnh hưởng tới nhiều hàng có thể đi tới các hàng theo bất kỳ trật tự nào.

- Khi một trigger sau mức hàng được chạy, tất cả các thay đổi dữ liệu được lệnh bên ngoài thực hiện là hoàn tất rồi, và là nhìn thấy được đối với hàm trigger được triệu gọi.

Nếu hàm trigger của bạn được viết trong bất kỳ ngôn ngữ thủ tục tiêu chuẩn nào, thì các lệnh ở trên chỉ áp dụng nếu hàm đó được khai báo VOLATILE. Các hàm được khai báo STABLE hoặc IMMUTABLE sẽ không thấy các thay đổi được lệnh gọi đó thực hiện trong bất kỳ trường hợp nào.

Thông tin thêm về các qui định khả năng nhìn thấy dữ liệu có thể được thấy trong Phần 43.4. Ví dụ trong Phần 36.4 có một trình diễn các qui tắc đó.

36.3. Viết các hàm trigger trong C

Phần này mô tả các chi tiết mức thấp của giao diện cho một hàm trigger. Thông tin này chỉ cần thiết khi viết các hàm trigger trong C. Nếu bạn đang sử dụng một ngôn ngữ mức cao hơn thì các chi tiết đó được xử trí cho bạn. Trong hầu hết các trường hợp bạn nên xem xét việc sử dụng một ngôn ngữ thủ tục trước khi viết các trigger của bạn trong C. Tài liệu về từng ngôn ngữ thủ tục giải thích cách để viết một trigger trong ngôn ngữ đó.

Các hàm trigger phải sử dụng giao diện của trình quản lý hàm "phiên bản 1".

Khi một hàm được trình quản lý trigger gọi, nó không được truyền bất kỳ đối số thông thường nào, mà nó được truyền một con trỏ "ngữ cảnh" trỏ một cấu trúc TriggerData. Các hàm C có thể kiểm tra liệu chúng đã được gọi từ trình quản lý trigger hay chưa bằng việc thực thi macro:

```
CALLED_AS_TRIGGER(fcinfo)
```

mà mở rộng tới:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Nếu điều này trả về đúng (true), thì là an toàn để đưa ra fcinfo->context to type TriggerData * và chắc chắn về cấu trúc được trỏ tới TriggerData. Hàm đó phải không tùy biến cấu trúc TriggerData hoặc bất kỳ dữ liệu nào nó trỏ tới.

struct TriggerData được xác định trong commands/trigger.h:

```
typedef struct TriggerData
{
        NodeTag type;
        TriggerEvent tg_event;
        Relation tg_relation;
        HeapTuple tg_trigtuple;
        HeapTuple tg_newtuple;
        Trigger *tg_trigger;
        Buffer tg_trigtuplebuf;
        Buffer tg_newtuplebuf;
} TriggerData;
```

trong đó các thành viên được xác định như sau:

type

Luôn là T_TriggerData.

tg_event

Mô tả sự kiện theo đó hàm được gọi. Bạn có thể sử dụng các macro sau để kiểm tra tg_event:

TRIGGER_FIRED_BEFORE(tg_event)

Trả về đúng (true) nếu trigger được chạy trước hoạt động đó.

TRIGGER_FIRED_AFTER(tg_event)

Trả về đúng (true) nếu trigger đó được chạy sau hoạt động đó.

TRIGGER_FIRED_FOR_ROW(tg_event)

Trả về đúng (true) nếu trigger đó được chạy cho một sự kiện mức hàng.

TRIGGER_FIRED_FOR_STATEMENT(tg_event)

Trả về đúng (true) nếu trigger đó được chạy cho một sự kiện mức lệnh.

TRIGGER_FIRED_BY_INSERT(tg_event)

Trả về đúng (true) nếu trigger đó đã được chạy bằng một lệnh INSERT.

TRIGGER_FIRED_BY_UPDATE(tg_event)

Trả về đúng (true) nếu trigger đó đã được chạy bằng một lệnh UPDATE.

TRIGGER_FIRED_BY_DELETE(tg_event)

Trả về đúng (true) nếu trigger đó đã được chạy bằng một lệnh DELETE.

TRIGGER_FIRED_BY_TRUNCATE(tg_event)

Trả về đúng (true) nếu trigger đó đã được chạy bằng một lệnh TRUNCATE.

tg_relation

Một con trỏ tới một cấu trúc mô tả quan hệ mà trigger đó được chạy cho. Hãy nhìn vào utils/rel.h để có các chi tiết về cấu trúc đó. Những điều thú vị nhất là tg_relation->rd_att (trình mô tả các bộ dữ liệu) và tg_relation->rd_rel->relname (tên quan hệ; dạng đó không là char* mà là NameData; hãy sử dụng SPI_getrelname(tg_relation) để có một char* nếu bạn cần một bản sao của tên đó).

tg_trigtuple

Một con trỏ tới hàng theo đó trigger đã được chạy. Điều này là hàng đang được chèn, được cập nhật, hoặc bị xóa. Nếu trigger này đã được chạy cho một lệnh INSERT hoặc DELETE thì điều này là những gì bạn sẽ trả về từ hàm nếu bạn không muốn thay thế hàng đó bằng một hàng khác (trong trường hợp của INSERT) hoặc bỏ qua hoạt động đó.

tg_newtuple

Một con trỏ tới phiên bản mới của hàng đó, nếu trigger đó đã được chạy cho một UPDATE, và NULL nếu nó là cho một lệnh INSERT hoặc DELETE. Điều này là những gì bạn phải trả về từ hàm đó nếu sự kiện đó là UPDATE và bạn không muốn thay thế hàng đó bằng một hàng khác hoặc bỏ qua hoạt động đó.

tg_trigger

Một con trỏ tới một cấu trúc của dạng Trigger, được xác định trong utils/rel.h:

```
typedef struct Trigger
{
```

```
        Oid tgoid;
        char *tgname;
        Oid tgfoid;
        int16 tgtype;
        char tgenabled;
        bool tgisinternal;
        Oid tgconstrrelid;
        Oid tgconstrindid;
        Oid tgconstraint;
        bool tgdeferrable;
        bool tginitdeferred;
        int16 tgnargs;
        int16 tgnattr;
        int16 *tgattr;
        char **tgargs;
        char *tgqual;
} Trigger;
```

trong đó tgname là tên của trigger, tgnargs là số các đối số trong tgargs, và tgargs là một mảng các con trỏ tới các đối số được chỉ định trong lệnh CREATE TRIGGER.

Các thành viên khác là chỉ để sử dụng trong nội bộ.

tg_trigtuplebuf

Bộ nhớ đệm chứa tg_trigtuple, hoặc InvalidBuffer nếu không có bộ dữ liệu nào như vậy hoặc nó không được lưu trữ trong một bộ nhớ đệm của đĩa.

tg_newtuplebuf

Bộ nhớ đệm chứa tg_newtuple, hoặc InvalidBuffer nếu không có bộ dữ liệu nào như vậy hoặc nó không được lưu trữ trong một bộ nhớ đệm của đĩa.

Một hàm trigger phải trả về hoặc một con trỏ HeapTuple hoặc một con trỏ NULL (không phải một giá trị null SQL, đó là, không thiết lập isNull đúng). Hãy thận trọng để trả về hoặc tg_trigtuple hoặc tg_newtuple, một cách phù hợp, nếu bạn không muốn sửa đổi hàng đang được vận hành trong đó.

34.4. Ví dụ trigger hoàn chỉnh

Đây là một ví dụ rất đơn giản về một hàm trigger được viết trong C. (Các ví dụ về các trigger được viết trong các ngôn ngữ thủ tục có thể thấy trong tài liệu của các ngôn ngữ thủ tục đó).

Hàm trigf nêu số các hàng trong bảng ttest và bỏ qua hoạt động thực tế nếu lệnh đó cố gắng chèn một giá trị null vào cột x. (Vì thế trigger đó hành động như một ràng buộc không null nhưng không bỏ giao dịch).

Trước nhất, định nghĩa bản:

```
CREATE TABLE ttest (
        x integer
);
```

Đây là mã nguồn của hàm trigger đó:

```
#include "postgres.h"
#include "executor/spi.h" /* this is what you need to work with SPI */
#include "commands/trigger.h" /* ... and triggers */
```

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
        TriggerData *trigdata = (TriggerData *) fcinfo->context;
        TupleDesc tupdesc;
        HeapTuple rettuple;
        char *when;
        bool checknull = false;
        bool isnull;
        int ret, i;

        /* make sure it's called as a trigger at all */
        if (!CALLED_AS_TRIGGER(fcinfo))
                elog(ERROR, "trigf: not called by trigger manager");

        /* tuple to return to executor */
        if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
                rettuple = trigdata->tg_newtuple;
        else
                rettuple = trigdata->tg_trigtuple;

        /* check for null values */
        if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
                && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
                checknull = true;

        if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
                when = "before";
        else
                when = "after ";

        tupdesc = trigdata->tg_relation->rd_att;

        /* connect to SPI manager */
        if ((ret = SPI_connect()) < 0)
                elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

        /* get number of rows in table */
        ret = SPI_exec("SELECT count(*) FROM ttest", 0);

        if (ret < 0)
                elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

        /* count(*) returns int8, so be careful to convert */
        i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                        SPI_tuptable->tupdesc,
                                        1,
                                        &isnull));

        elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

        SPI_finish();
```

```
        if (checknull)
        {
                SPI_getbinval(rettuple, tupdesc, 1, &isnull);
                if (isnull)
                        rettuple = NULL;
        }
        return PointerGetDatum(rettuple);
}
```

Sau khi bạn đã biên dịch mã nguồn (xem Phần 35.9.6), hãy khai báo hàm và các trigger đó:

```
CREATE FUNCTION trigf() RETURNS trigger
        AS 'filename'
        LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
        FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
        FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Bây giờ bạn có thể kiểm thử hoạt động của trigger đó:

```
=> INSERT INTO ttest VALUES (NULL);
INFO: trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired (Chèn bị bỏ qua và trigger AFTER không được chạy)
=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO: trigf (fired before): there are 0 rows in ttest
INFO: trigf (fired after ): there are 1 rows in ttest
                                     ^^^^^^^^^
                        remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;

 x
---
1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO: trigf (fired before): there are 1 rows in ttest
INFO: trigf (fired after ): there are 2 rows in ttest
                                     ^^^^^^^
                        remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
1
2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO: trigf (fired before): there are 2 rows in ttest
```

```
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO: trigf (fired before): there are 2 rows in ttest
INFO: trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
1
4
(2 rows)

=> DELETE FROM ttest;
INFO: trigf (fired before): there are 2 rows in ttest
INFO: trigf (fired before): there are 1 rows in ttest
INFO: trigf (fired after ): there are 0 rows in ttest
INFO: trigf (fired after ): there are 0 rows in ttest
                                        ^^^^^^
                               remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Có nhiều ví dụ phức tạp hơn trong src/test/regress/regress.c và trong contrib/spi.

Chương 37. Hệ thống quy tắc

Chương này thảo luận hệ thống quy tắc trong PostgreSQL. Các hệ thống quy tắc trong sản xuất về khái niệm là đơn giản, nhưng có nhiều điểm tế nhị có liên quan trong việc sử dụng thực sự chúng.

Vài hệ thống cơ sở dữ liệu khác định nghĩa các quy tắc cơ sở dữ liệu tích cực, chúng thường các thủ tục và các trigger được lưu trữ. Trong PostgreSQL, chúng cũng có thể được triển khai bằng việc sử dụng các hàm và các trigger.

Hệ thống quy tắc (nói chính xác hơn, hệ thống quy tắc viết lại các truy vấn) là hoàn toàn khác với các thủ tục và trigger được lưu trữ. Nó tùy biến các truy vấn với việc cân nhắc các quy tắc, và sau đó truyền truy vấn được tùy biến tới trình lên kế hoạch truy vấn để lên kế hoạch và thực thi. Nó là rất mạnh, và có thể được nhiều thứ sử dụng như các thủ tục ngôn ngữ truy vấn, các kiểu nhìn, và các phiên bản. Các thiết lập lý thuyết và sức mạnh của hệ thống quy tắc này cũng được thảo luận trong tài liệu *Về các Quy tắc, các Thủ tục, Lưu trữ và các Kiểu nhìn trong các Hệ thống Cơ sở dữ liệu và một Khung Thống nhất cho các Quy tắc Sản xuất có Sử dụng Mô hình hóa Phiên bản trong một Hệ thống Cơ sở dữ liệu* (*On Rules, Procedures, Caching and Views in Database Systems and A Unified Framework for Version Modeling Using Production Rules in a Database System*).

37.1. Cây Truy vấn

Để hiểu cách thức hệ thống quy tắc làm việc như thế nào, là cần thiết phải biết khi nào nó được triệu gọi và đầu vào và các kết quả đầu ra của nó là gì.

Hệ thống quy tắc được đặt giữa trình phân tích cú pháp (parser) và trình lên kế hoạch (planner). Nó lấy đầu ra của trình phân tích cú pháp, một cây truy vấn, và các quy tắc viết lại do người sử dụng định nghĩa, chúng cũng là các cây truy vấn với vài thông tin thêm, và tạo ra zero và nhiều cây truy vấn hơn như là kết quả. Vì thế đầu vào và đầu ra của nó luôn là những điều mà bản thân trình phân tích cú pháp có thể đã tạo ra và vì thế, bất kỳ điều gì nó nhìn thấy, về cơ bản, có thể được miêu tả như một lệnh SQL.

Cây truy vấn là cái gì? Nó là một sự đại diện của một lệnh SQL, trong đó các phần đơn nhất được xây dựng được lưu giữ một cách tách biệt. Các cây truy vấn đó có thể được trình bày trong lưu ký máy chủ nếu bạn thiết lập các tham số cấu hình `debug_print_parse`, `debug_print_rewritten`, hoặc `debug_print_plan`. Các hành động theo quy tắc cũng được lưu trữ như là các cây truy vấn, trong catalog hệ thống `pg_rewrite`. Chúng không được định dạng như là đầu ra lưu ký, mà chúng chính xác có thông tin y hệt.

Việc đọc một cây truy vấn thông thường đòi hỏi vài kinh nghiệm. Nhưng vì các đại diện cây truy vấn của SQL là đủ để hiểu hệ thống quy tắc, chương này sẽ không dạy cách để đạt được chúng. Khi đọc các đại diện SQL của các cây truy vấn trong chương này, là cần thiết để có khả năng nhận diện các phần mà lệnh bị đổ vỡ trong đó khi nó ở trong cấu trúc cây truy vấn. Các phần của một cây truy vấn là

dạng lệnh

Đây là giá trị đơn giản nói lệnh nào (SELECT, INSERT, UPDATE, DELETE) đã tạo ra cây truy vấn.

bảng dãy

Bảng dãy là danh sách các mối quan hệ được sử dụng trong truy vấn. Trong một lệnh SELECT có các mối quan hệ được đưa ra sau từ khóa FROM.

Mỗi khoản đầu vào của bảng dãy nhận diện một bảng hoặc kiểu nhìn và nói theo tên nào nó được gọi trong các phần khác của truy vấn đó. Trong cây truy vấn, các khoản đầu vào của bảng dãy được tham chiếu bằng số hơn là bằng tên, vì thế ở đây không là vấn đề gì nếu có các tên trùng lắp như điều có thể có trong một lệnh SQL. Điều này có thể xảy ra sau khi các bảng dãy các quy tắc đã được sát nhập vào. Các ví dụ trong chương này sẽ không có tình huống này.

quan hệ kết quả

Đây là chỉ số trong bảng dãy nhận diện quan hệ trong đó có các kết quả của truy vấn.

Các truy vấn SELECT thường không có quan hệ kết quả. Trường hợp đặc biệt của SELECT INTO là hầu như giống y hệt với một CREATE TABLE theo sau là INSERT ... SELECT và không được thảo luận riêng rẽ ở đây.

Đối với các lệnh INSERT, UPDATE, và DELETE, quan hệ kết quả là bảng (hoặc kiểu nhìn!), trong đó các thay đổi sẽ có tác động.

danh sách mục tiêu

Danh sách mục tiêu là một danh sách các biểu thức xác định kết quả của truy vấn đó. Trong trường hợp của một lệnh SELECT, các biểu thức đó là các biểu thức xây dựng đầu ra cuối cùng của truy vấn. Chúng tương ứng với các biểu thức giữa các từ khóa SELECT và FROM. (* chỉ là một sự viết tắt cho tất cả các tên cột của một quan hệ. Nó được trình phân tích cú pháp mở rộng trong các cột riêng rẽ, nên hệ thống quy tắc không bao giờ nhìn thấy nó).

Các lệnh DELETE không cần một danh sách đích vì chúng không tạo ra bất kỳ kết quả nào. Trong thực tế, trình lên kế hoạch sẽ thêm một khoản đầu vào đặc biệt CTID vào danh sách đích rỗng đó, nhưng điều này là sau hệ thống quy tắc và sẽ được thảo luận sau; đối với hệ thống quy tắc, danh sách đích là rỗng.

Đối với lệnh INSERT, danh sách đích mô tả các hàng mới sẽ đi vào quan hệ kết quả. Nó bao gồm các biểu thức trong mệnh đề VALUES hoặc các biểu thức từ mệnh đề SELECT trong INSERT ... SELECT. Bước đầu của qui trình viết lại sẽ thêm các khoản đầu vào của danh sách đích bất kỳ cột nào từng chưa được lệnh gốc ban đầu chỉ định mà có các mặc định. Bất kỳ cột còn lại nào (hoặc không với một giá trị được đưa ra, hoặc không có một mặc định) sẽ được trình lên kế hoạch điền bằng một biểu thức hằng số null.

Đối với các lệnh UPDATE, danh sách đích mô tả các hàng mới sẽ thay thế các hàng cũ. Trong hệ thống quy tắc, nó gồm chỉ các biểu thức từ phần SET column = expression của lệnh đó. Trình lên kế hoạch sẽ điều khiển các cột còn thiếu bằng việc chèn các biểu thức sao chép các

dữ liệu từ hàng cũ vào hàng mới. Và nó sẽ thêm khoản đầu vào đặc biệt CTID cũng như đối với DELETE.

Mỗi khoản đầu vào trong danh sách đích gồm một biểu thức có thể là một giá trị hằng, một biến trỏ tới một cột của một trong các quan hệ trong bảng dãy, một tham số, hoặc một cây biểu thức được tạo ra từ các lời gọi hàm, các hằng, các biến và các toán tử, …

định tính

Định tính của truy vấn là một biểu thức rất giống biểu thức của những thứ có trong các khoản đầu vào của danh sách đích. Giá trị kết quả của biểu thức này là một Boolean nói liệu hoạt động đó (hoặc) đối với hàng kết quả cuối cùng sẽ được thực thi hay không. Nó tương ứng với mệnh đề WHERE của lệnh SQL.

cây liên kết

Cây liên kết của truy vấn chỉ ra cấu trúc của mệnh để FROM. Đối với một truy vấn đơn giản như SELECT ... FROM a, b, c, thì cây liên kết chỉ là một danh sách các khoản FROM, vì chúng ta được phép liên kết chúng theo bất kỳ trật tự nào. Nhưng khi các biểu thức JOIN, đặc biệt là các liên kết ngoài, được sử dụng, thì chúng ta phải liên kết theo trật tự được các liên kết đó chỉ ra. Trong trường hợp đó, cây liên kết chỉ ra cấu trúc của các biểu thức JOIN. Các giới hạn có liên quan tới các mệnh đề đặc biệt JOIN (từ các biểu thức ON hoặc USING) sẽ được lưu trữ như là các biểu thức định tính được gắn tới các nút của cây liên kết đó. Hóa ra là cũng rất thuận tiện để lưu trữ biểu thức WHERE mức đỉnh như một định tính được gắn tới khoản của cây liên kết mức đỉnh.

những điều khác

Các phần khác của cây truy vấn giống như mệnh đề ORDER BY sẽ không thú vị ở đây. Hệ thống quy tắc thay thế cho vài khoản đầu vào ở đó khi áp dụng các quy tắc, nhưng không có nhiều điều phải làm với những phần cơ bản của hệ thống quy tắc.

37.2. Kiểu nhìn và hệ thống quy tắc

Các kiểu nhìn trong PostgreSQL được triển khai bằng việc sử dụng hệ thống quy tắc. Trên thực tế, về cơ bản không có sự khác biệt giữa:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

khi được so sánh với 2 lệnh:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

vì điều này chính xác là những gì lệnh CREATE VIEW thực hiện trong nội bộ. Điều này có vài hiệu ứng phụ. Một trong số chúng là thông tin về một kiểu nhìn trong các catalog hệ thống của PostgreSQL là chính xác y hệt như nó dành cho bảng. Vì thế đối với trình phân tích cú pháp, tuyệt đối không có sự khác biệt nào giữa bảng và kiểu nhìn. Chúng là thứ y hệt nhau: các quan hệ.

37.2.1. Quy tắc SELECT làm việc thế nào

Các quy tắc ON SELECT được áp dụng cho tất cả các truy vấn như là bước cuối cùng, thậm chí nếu lệnh được đưa ra đó là INSERT, UPDATE hoặc DELETE. Và chúng có các ngữ nghĩa khác nhau từ các quy tắc trong các dạng lệnh khác theo đó chúng tùy biến cây truy vấn tại chỗ thay vì việc tạo một cây mới. Vì thế các quy tắc SELECT được mô tả trước nhất.

Hiện hành, chỉ có một hành động trong một quy tắc ON SELECT, và nó phải là một hành động SELECT không có điều kiện đó là INSTEAD. Giới hạn này đã được yêu cầu để làm cho các quy tắc an toàn đủ để mở chúng cho những người sử dụng thông thường, và nó giới hạn các quy tắc ON SELECT để hành động giống như các kiểu nhìn (view).

Các ví dụ cho chương này là 2 kiểu nhìn liên kết thực hiện vài tính toán và vài kiểu nhìn nữa sử dụng chúng. Một trong 2 kiểu nhìn đầu tiên được tùy biến sau đó bằng việc bổ sung thêm các quy tắc cho các hành động INSERT, UPDATE, và DELETE sao cho kết quả cuối cùng sẽ là một kiểu nhìn mà hành xử giống như một bảng thực tế với vài chức năng thú vị. Đây không phải là một ví dụ đơn giản như vậy để bắt đầu và điều này làm cho mọi điều khó khăn hơn để tiến hành. Nhưng là tốt hơn để có một ví dụ bao trùm tất cả các điểm được thảo luận từng bước một hơn là việc có nhiều ví dụ khác nhau mà có thể trộn lẫn trong đầu.

Ví dụ, chúng ta cần một ít hàm min trả về các giá trị 2 số nguyên nhỏ hơn. Tạo điều đó như là:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
        SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

Các bảng thực tế chúng ta cần trong 2 mô tả hệ thống quy tắc đầu tiên là thế này:

```
CREATE TABLE shoe_data (
        shoename text, -- primary key
        sh_avail integer, -- available number of pairs
        slcolor text, -- preferred shoelace color
        slminlen real, -- minimum shoelace length
        slmaxlen real, -- maximum shoelace length
        slunit text -- length unit
);

CREATE TABLE shoelace_data (
        sl_name text, -- primary key
        sl_avail integer, -- available number of pairs
        sl_color text, -- shoelace color
        sl_len real, -- shoelace length
        sl_unit text -- length unit
);

CREATE TABLE unit (
        un_name text, -- primary key
        un_fact real -- factor to transform to cm
);
```

Như bạn có thể thấy, chúng thể hiện các dữ liệu của cửa hàng giầy dép.

Các kiểu nhìn được tạo ra như:

```
CREATE VIEW shoe AS
        SELECT sh.shoename,
                sh.sh_avail,
```

```
                sh.slcolor,
                sh.slminlen,
                sh.slminlen * un.un_fact AS slminlen_cm,
                sh.slmaxlen,
                sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                sh.slunit
          FROM shoe_data sh, unit un
         WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
        SELECT s.sl_name,
                s.sl_avail,
                s.sl_color,
                s.sl_len,
                s.sl_unit,
                s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
        SELECT rsh.shoename,
                rsh.sh_avail,
                rsl.sl_name,
                rsl.sl_avail,
                min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

Lệnh CREATE VIEW đối với kiểu nhìn shoelace (nó là kiểu nhìn đơn giản nhất chúng ta có) sẽ tạo một quan hệ shoelace và một khoản đầu vào trong pg_rewrite mà nói rằng có một quy tắc được viết lại phải được áp dụng bất kỳ khi nào quan hệ shoelace được tham chiếu trong bảng dãy của một truy vấn. Quy tắc đó không có định tính quy tắc (được thảo luận sau, với các quy tắc không SELECT, vì các quy tắc SELECT hiện không thể có chúng) và nó là INSTEAD. Lưu ý rằng các định tính quy tắc không là y hệt như các định tính truy vấn. Hành động ngoài quy tắc có một định tính truy vấn. Hành động của quy tắc là cây truy vấn mà là một bản sao của lệnh SELECT trong lệnh tạo kiểu nhìn đó.

> **Lưu ý**: 2 khoản đầu vào bảng dãy thêm cho NEW và OLD mà bạn có thể thấy trong khoản đầu vào pg_rewrite entry sẽ không thú vị đối với các quy tắc SELECT.

Bây giờ chúng ta tới unit, shoe_data và shoelace_data và chạy truy vấn đơn giản trong kiểu nhìn:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
```

```
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
------------+----------+-----------+--------+----------+-----------
sl1         |        5 | black    |     80 | cm       |        80
sl2         |        6 | black    |    100 | cm       |       100
sl7         |        7 | brown    |     60 | cm       |        60
sl3         |        0 | black    |     35 | inch     |      88.9
sl4         |        8 | black    |     40 | inch     |     101.6
sl8         |        1 | brown    |     40 | inch     |     101.6
sl5         |        4 | brown    |      1 | m        |       100
sl6         |        0 | brown    |    0.9 | m        |        90
(8 rows)
```

Đây là SELECT đơn giản nhất mà bạn có thể làm trong các kiểu nhìn của bạn, nên chúng ta nhân cơ hội này để giải thích những điều cơ bản của các quy tắc kiểu nhìn. SELECT * FROM shoelace từng được diễn giải bằng trình phân tích cú pháp và đã tạo ra cây truy vấn:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

và điều này được đưa ra cho hệ thống quy tắc. Hệ thống quy tắc đi qua bảng dãy và kiểm tra liệu có các quy tắc cho bất kỳ quan hệ nào hay không. Khi xử lý khoản đầu vào bảng dãy cho shoelace (thứ duy nhất cho tới nay) thì nó thấy quy tắc _RETURN với cây truy vấn:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

Để mở rộng kiểu nhìn đó, trình viết lại đơn giản tạo ra một khoản đầu vào bảng dãy của truy vấn con có chứa cây truy vấn hành động của quy tắc đó, và thay thế khoản đầu vào bảng dãy này đối với khoản đầu vào gốc ban đầu mà đã tham chiếu tới kiểu nhìn đó.

Kết quả của cây truy vấn được viết lại hầu như là y hệt dường như bạn đã gõ:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

Tuy nhiên, có một sự khác biệt: bảng dãy của truy vấn con có 2 khoản đầu vào thêm shoelace old và shoelace new. Các khoản đầu vào đó không tham gia trực tiếp trong truy vấn, vì chúng không được

tham chiếu bởi cây liên kết hoặc danh sách đích của truy vấn con đó. Trình viết lại sử dụng chúng để lưu trữ thông tin kiểm tra quyền ưu tiên truy cập từng ban đầu được trình bày trong khoản đầu vào của bảng dãy mà đã tham chiếu tới kiểu nhìn đó. Theo cách này, trình thực thi vẫn sẽ kiểm tra xem người sử dụng có các quyền ưu tiên đúng phù hợp để truy cập kiểu nhìn đó hay không, thậm chí dù không có sử dụng trực tiếp kiểu nhìn đó trong truy vấn được viết lại.

Đó là quy tắc đầu tiên được áp dụng. Hệ thống quy tắc sẽ tiếp tục việc kiểm tra các khoản đầu vào còn lại của bảng dãy trong truy vấn đỉnh (trong ví dụ này không có hơn), và nó sẽ kiểm tra một cách đệ quy các khoản đầu vào của bảng dãy trong truy vấn con được thêm vào để xem liệu có bất kỳ khoản đầu vào nào tham chiếu tới các kiểu nhìn tham chiếu hay không. (Nhưng nó sẽ không mở rộng old hoặc new - nếu không thì chúng ta có thể có sự đệ quy bất tận!). Trong ví dụ này, không có các quy tắc viết lại cho shoelace_data hoặc unit, vì thế việc viết lại là hoàn tất và thứ ở trên là kết quả cuối cùng được đưa ra cho trình lên kế hoạch.

Bây giờ chúng ta muốn viết truy vấn tìm ra theo đó các đôi giày hiện trong kho chúng ta có các dây giày phù hợp (về màu sắc và độ dài) và nơi mà tổng số các đôi khớp chính xác là >= 2.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
sh1      |        2 | sl1     |        5 |           2
sh3      |        4 | sl7     |        7 |           4
(2 rows)

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

Quy tắc đầu được áp dụng sẽ là quy tắc cho kiểu nhìn shoe_ready và nó đưa ra cây truy vấn:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

Tương tự, các quy tắc cho giầy và dây giầy được thay thế trong bảng dãy của truy vấn con, dẫn tới cây truy vấn cuối cùng với 3 mức:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
```

```
                rsl.sl_avail,
                min(rsh.sh_avail, rsl.sl_avail) AS total_avail
           FROM (SELECT sh.shoename,
                        sh.sh_avail,
                        sh.slcolor,
                        sh.slminlen,
                        sh.slminlen * un.un_fact AS slminlen_cm,
                        sh.slmaxlen,
                        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                        sh.slunit
                   FROM shoe_data sh, unit un
                  WHERE sh.slunit = un.un_name) rsh,
                 (SELECT s.sl_name,
                         s.sl_avail,
                         s.sl_color,
                         s.sl_len,
                         s.sl_unit,
                         s.sl_len * u.un_fact AS sl_len_cm
                    FROM shoelace_data s, unit u
                   WHERE s.sl_unit = u.un_name) rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail > 2;
```

Hóa ra là trình lên kế hoạch sẽ ép cho cây này thành một cây truy vấn có 2 mức: các lệnh SELECT ở dưới cùng sẽ "được kéo lên" thành lệnh SELECT ở giữa vì không cần phải xử lý chúng riêng rẽ.

Nhưng SELECT ở giữa sẽ vẫn tách bạch khỏi đỉnh, vì nó chứa các hàm tính tổng. Nếu chúng ta đã kéo chúng lên thì nó có thể thay đổi hành vi của SELECT ở trên cùng, điều chúng ta không muốn. Tuy nhiên, việc ép cây truy vấn là sự tối ưu hóa mà hệ thống viết lại không phải lo cho bản thân nó.

37.2.2. Quy tắc kiểu nhìn trong các lệnh không SELECT

2 chi tiết của cây truy vấn sẽ không được động chạm tới trong mô tả các quy tắc kiểu nhìn ở trên. Chúng là dạng lệnh và quan hệ kết quả. Trên thực tế, các quy tắc kiểu nhìn không cần thông tin này.

Chỉ có ít khác biệt giữa một cây truy vấn đối với một lệnh SELECT và một sự khác biệt cho bất kỳ lệnh nào khác. Rõ ràng, chúng có một dạng lệnh khác và đối với một lệnh khác SELECT, quan hệ kết quả chỉ tới khoản đầu vào bảng dãy trong đó kết quả sẽ tới. Tất cả những điều khác là tuyệt đối y hệt. Vì thế, việc có 2 bảng t1 và t2 với các cột a và b, các cây truy vấn cho 2 lệnh đó là:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

là gần như giống hệt nhau. Đặc biệt:

- Các bảng dãy chứa các khoản đầu vào cho các bảng t1 và t2.
- Các danh sách đích có một biến trỏ tới cột b của khoản đầu vào bảng dãy đối với t2.
- Các biểu thức định tính so sánh các cột a của cả các khoản đầu vào bảng dãy về sự ngang bằng nhau.
- Các cây liên két chỉ ra một liên kết đơn giản giữa t1 và t2.

Hệ quả là, cả 2 cây truy vấn đó làm ra các kế hoạch thực thi tương tự nhau: Chúng cả 2 liên kết qua

2 bảng đó. Đối với lệnh UPDATE thì các cột thiếu từ t1 được trình lên kế hoạch thêm vào danh sách đích và cây truy vấn sẽ được đọc như là:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

và vì thế trình thực thi chạy qua liên kết đó sẽ tạo ra chính xác tập kết quả y hệt như:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

sẽ làm. Nhưng có một chút vấn đề trong UPDATE: Trình thực thi không quan tâm các kết quả từ liên kết mà nó đang làm sẽ có ý nghĩa gì. Nó chỉ tạo ra tập hợp các hàng kết quả. Sự khác biệt là, một thứ là một lệnh SELECT và thứ kia là một lệnh UPDATE được điều khiển trong trình gọi của trình thực thi. Trình gọi vẫn biết (nhìn vào cây truy vấn) rằng đây là một lệnh UPDATE, và nó biết rằng kết quả này sẽ đi vào bảng t1. Nhưng hàng nào trong các hàng ở đó phải bị hàng mới thay thế?

Để giải quyết vấn đề này, một khoản đầu vào khác được thêm vào danh sách đích trong các lệnh UPDATE (và cũng trong DELETE): mã ID bộ dữ liệu hiện hành (CTID). Đây là cột hệ thống có chứa số và vị trí khối tệp trong khối dành cho hàng đó. Biết bảng đó, CTID có thể được sử dụng để truy xuất hàng ban đầu của t1 sẽ được cập nhật. Sau khi thêm CTID vào danh sách đích, truy vấn đó thực sự trông giống như:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Bây giờ một chi tiết khác của PostgreSQL tham gia vào giai đoạn này. Các hàng của bảng cũ sẽ không bị ghi đè, và điều này giải thích vì sao ROLLBACK là nhanh. Trong một UPDATE, hàng kết quả mới được chèn vào trong bảng (sau việc bóc được CTID) và trong đầu đề hàng của hàng cũ mà CTID được trỏ tới, các khoản đầu vào cmax và xmax sẽ được thiết lập cho bộ đếm lệnh hiện hành và ID giao dịch hiện hành. Vì thế hàng cũ được ẩn đi, và sau khi giao dịch thực thi trình làm sạch chân không có thể thực sự loại bỏ nó.

Biết tất cả điều đó, chúng ta có thể đơn giản áp dụng các quy tắc kiểu nhìn theo cách thức tuyệt đối y hệt với bất kỳ lệnh nào. Sẽ không có sự khác biệt nào cả.

37.2.3. Sức mạnh các kiểu nhìn trong PostgreSQL

Điều ở trên trình bày cách thức hệ thống quy tắc kết hợp các định nghĩa kiểu nhìn trong cây truy vấn gốc. Trong ví dụ thứ 2, một lệnh SELECT đơn giản từ một kiểu nhìn đã tạo ra một cây truy vấn cuối cùng là một liên kết của 4 bảng (unit đã được sử dụng 2 lần với các tên khác nhau).

Lợi ích của việc triển khai các kiểu nhìn với hệ thống quy tắc là, trình lên kế hoạch có tất cả thông tin về các bảng nào phải được quét cộng với các mối quan hệ giữa các bảng đó, cộng với các định tính có giới hạn từ các kiểu nhìn đó, cộng với các định tính từ truy vấn gốc ban đầu trong một cây truy vấn duy nhất. Và đây còn là tình huống khi mà truy vấn gốc ban đầu là một liên kết đối với các kiểu nhìn rồi. Trình lên kế hoạch phải quyết định đâu là con đường tốt nhất để thực thi truy vấn đó, và càng nhiều thông tin trình lên kế hoạch có, quyết định này có thể là càng tốt hơn. Và hệ thống quy tắc như được triển khai đó trong PostgreSQL sẽ đảm bảo rằng đây là tất cả thông tin có sẵn về truy vấn tới thời điểm đó.

37.2.4. Cập nhật một kiểu nhìn

Điều gì xảy ra nếu một kiểu nhìn được đặt tên như là quan hệ đích đối với lệnh INSERT, UPDATE,

hoặc DELETE? Sau khi tiến hành các thay thế được mô tả ở trên, chúng ta sẽ có một cây truy vấn trong đó quan hệ kết quả trỏ vào một khoản đầu vào bảng dãy của truy vấn con. Điều này sẽ không làm việc, nên trình viết lại sẽ đưa ra một lỗi nếu nó thấy một thứ như vậy đã được tạo ra.

Để thay đổi điều này, chúng ta có thể định nghĩa các quy tắc mà sửa đổi hành vi đó của các dạng lệnh. Đây là chủ đề của phần tiếp sau.

37.3. Qui tắc về INSERT, UPDATE, và DELETE

Các quy tắc được định nghĩa về INSERT, UPDATE, và DELETE là khác nhau đáng kể với quy tắc kiểu nhìn được mô tả trong phần trước. Trước nhất, lệnh CREATE RULE của chúng cho phép nhiều hơn:

- Chúng được phép không có hành động nào.
- Chúng có thể có nhiều hành động.
- Chúng có thể là INSTEAD hoặc ALSO (mặc định).
- Các quan hệ giả NEW và OLD trở thành có ích.
- Chúng có thể có các định tính quy tắc.

Thứ 2, chúng không sửa đổi cây truy vấn tại chỗ. Thay vào đó chúng tạo ra zero hoặc nhiều cây truy vấn mới hơn và có thể bỏ đi cây truy vấn gốc ban đầu.

37.3.1. Quy tắc cập nhật làm việc thế nào

Giữ cú pháp:

```
CREATE [ OR REPLACE ] RULE name AS ON event
        TO table [ WHERE condition ]
        DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

trong đầu. Sau đó, hãy *cập nhật các quy tắc*, nghĩa là các quy tắc được xác định trong INSERT, UPDATE, hoặc DELETE.

Các quy tắc cập nhật được hệ thống quy tắc áp dụng khi quan hệ kết quả và dạng lệnh của một cây truy vấn là ngang nhau đối với đối tượng và sự kiện được đưa ra trong lệnh CREATE RULE. Đối với các quy tắc cập nhật, hệ thống quy tắc tạo ra một danh sách các cây truy vấn. Ban đầu danh sách cây truy vấn là rỗng. Có thể là zero (từ khóa NOTHING), là 1, hoặc nhiều ứng dụng. Để đơn giản hóa, chúng ta sẽ xem xét một quy tắc với 1 hành động. Quy tắc này có thể có một sự định tính hoặc không và nó có thể là INSTEAD hoặc ALSO (mặc định).

Quy tắc định tính là gì? Đây là sự hạn chế nói lên khi nào thì các hành động của quy tắc sẽ được thực hiện và khi nào không được thực hiện. Sự định tính này chỉ có thể tham chiếu tới các quan hệ giả NEW và/hoặc OLD, chúng về cơ bản đại diện cho quan hệ đã được đưa ra như là đối tượng (nhưng với một ý nghĩa đặc biệt).

Vì thế chúng ta có 3 trường hợp tạo ra 3 cây truy vấn sau cho quy tắc 1 hành động.

Không định tính, với cây truy vấn hoặc ALSO hoặc INSTEAD

cây truy vấn từ hành động quy tắc với định tính của cây truy vấn gốc ban đầu được thêm vào

Định tính được đưa ra và ALSO

cây truy vấn từ hành động quy tắc với định tính quy tắc và định tính của cây truy vấn gốc ban đầu được thêm vào

Định tính được đưa ra và INSTEAD

cây truy vấn từ hành động quy tắc với định tính quy tắc và định tính của cây truy vấn gốc ban đầu; và cây truy vấn gốc ban đầu với định tính quy tắc bị phủ định được thêm vào

Cuối cùng, nếu quy tắc đó là ALSO, thì cây truy vấn gốc ban đầu không thay đổi được thêm vào danh sách. Vì chỉ các quy tắc INSTEAD được định tính thêm vào rồi cây truy vấn gốc ban đầu, chúng ta kết thúc với hoặc 1 hoặc 2 cây truy vấn đầu ra cho một quy tắc với 1 hành động.

Đối với các quy tắc ON INSERT, truy vấn gốc ban đầu (nếu được INSTEAD kìm nén lại) được thực hiện trước khi bất kỳ hành động nào được các quy tắc thêm vào. Điều này cho phép các hành động thấy được (các) hàng được chèn vào. Nhưng đối với các quy tắc ON UPDATE và ON DELETE, truy vấn gốc ban đầu được thực hiện sau các hành động được các quy tắc thêm vào. Điều này đảm bảo rằng các hành động có thể thấy các hàng sẽ được cập nhật hoặc sẽ bị xóa; nếu không, các hành động đó có thể không làm gì vì chúng thấy không hàng nào khớp với các định tính của chúng.

Các cây truy vấn được sinh ra từ các hành động quy tắc được đưa vào hệ thống viết lại một lần nữa, và có thể nhiều quy tắc hơn được áp dụng tạo ra ít nhiều các cây truy vấn. Vì thế các hành động của một quy tắc phải có hoặc một dạng lệnh khác hoặc một quan hệ kết quả khác so với bản thân quy tắc hoạt động đó, nếu không thì qui trình đệ qui này sẽ kết thúc trong một vòng lặp bất tận. (Mở rộng đệ quy của một quy tắc sẽ được tìm thấy và được thông báo như một lỗi).

Các cây truy vấn được tìm thấy trong các hành động của catalog hệ thống pg_rewrite chỉ là các mẫu template. Vì chúng có thể tham chiếu tới các khoản đầu vào của bảng dãy đối với NEW và OLD, vài sự thay thế phải được thực hiện trước khi chúng có thể được sử dụng. Đối với bất kỳ tham chiếu nào đối với NEW, danh sách đích của truy vấn gốc ban đầu được tìm cho khoản đầu vào tương ứng. Nếu tìm thấy, biểu thức của khoản đầu vào đó sẽ thay thế tham chiếu đó. Nếu không, NEW có nghĩa là y hệt như OLD (đối với một UPDATE) hoặc được một giá trị null thay thế (đối với INSERT). Bất kỳ tham chiếu nào tới OLD được thay thế bằng một tham chiếu tới một khoản đầu vào bảng dãy, nó là quan hệ kết quả.

Sau khi hệ thống thực hiện xong việc áp dụng các quy tắc cập nhật, nó áp dụng các quy tắc kiểu nhìn cho (các) cây truy vấn được tạo ra.

Các kiểu nhìn không thể chèn các hành động cập nhật mới vì thế không có nhu cầu áp dụng các quy tắc cập nhật cho đầu ra của việc viết lại kiểu nhìn.

37.3.1.1. Quy tắc thứ nhất từng bước một

Chúng ta muốn theo dõi các thay đổi đối với cột sl_avail trong quan hệ shoelace_data. Vì thế chúng ta thiết lập một bảng lưu ký và một quy tắc mà theo điều kiện ghi một khoản đầu vào lưu ký khi một UPDATE được thực hiện trong shoelace_data.

```
CREATE TABLE shoelace_log (
        sl_name text, -- shoelace changed
        sl_avail integer, -- new available value
        log_who text, -- who did it
        log_when timestamp -- when
);
CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
        WHERE NEW.sl_avail <> OLD.sl_avail
        DO INSERT INTO shoelace_log VALUES (
                                                NEW.sl_name,
                                                NEW.sl_avail,
                                                current_user,
                                                current_timestamp
                                        );
```

Bây giờ ai đó làm:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

và chúng ta xem xét bảng lưu ký:

```
SELECT * FROM shoelace_log;

sl_name | sl_avail | log_who | log_when
----------+----------+----------+--------------------------------------------------
sl7        |        6 | Al         | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Đó là những gì chúng ta đã kỳ vọng. Những gì xảy ra ở phần nền tảng là sau đây. Trình phân tích cú pháp đã tạo ra cây truy vấn:

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

Có một quy tắc log_shoelace mà là ON UPDATE với biểu thức định tính quy tắc đó:

```
NEW.sl_avail <> OLD.sl_avail
```

và hành động:

```
INSERT INTO shoelace_log VALUES (
        new.sl_name, new.sl_avail,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

(Điều này trông hơi lạ vì bạn không thể bình thường viết INSERT ... VALUES ... FROM. Mệnh đề FROM ở đây chỉ để chỉ ra rằng có các khoản đầu vào bảng dãy trong cây truy vấn đối với new và old. Chúng là cần thiết sao cho chúng có thể được tham chiếu bằng các biến trong cây truy vấn của lệnh INSERT).

Quy tắc đó là một quy tắc ALSO được định tính, nên hệ thống quy tắc phải trả về 2 cây truy vấn: hành động quy tắc được sửa đổi và cây truy vấn gốc ban đầu. Trong bước 1, bảng dãy của truy vấn gốc ban đầu được kết hợp trong cây truy vấn hành động của quy tắc đó. Điều này gây ra:

```
INSERT INTO shoelace_log VALUES (
        new.sl_name, new.sl_avail,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
        shoelace_data shoelace_data;
```

Trong bước 2, định tính quy tắc được thêm vào nó, nên tập kết quả được giới hạn đối với các hàng nơi mà sl_avail thay đổi:

```
INSERT INTO shoelace_log VALUES (
        new.sl_name, new.sl_avail,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
        shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail;
```

(Điều này trông còn lạ hơn, vì INSERT ... VALUES cũng không có mệnh đề WHERE, mà trình lên kế hoạch và trình thực thi sẽ không khó khăn với nó. Chúng cần phải hỗ trợ chức năng y hệt này bất kỳ cách gì cho INSERT ... SELECT).

Trong bước 3, định tính của cây truy vấn gốc ban đầu được thêm vào, làm giới hạn tập kết quả xa hơn đối với chỉ các hàng có thể từng được truy vấn gốc ban đầu động chạm tới:

```
INSERT INTO shoelace_log VALUES (
        new.sl_name, new.sl_avail,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
        shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Bước 4 thay thế các tham chiếu cho NEW bằng các khoản đầu vào danh sách đích từ cây truy vấn gốc ban đầu hoặc bằng việc khớp các tham chiếu biến từ quan hệ kết quả:

```
INSERT INTO shoelace_log VALUES (
        shoelace_data.sl_name, 6,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
        shoelace_data shoelace_data
WHERE 6 <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

Bước 5 thay đổi các tham chiếu OLD thành các tham chiếu quan hệ kết quả:

```
INSERT INTO shoelace_log VALUES (
        shoelace_data.sl_name, 6,
        current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
        shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

Thế đấy. Vì quy tắc đó là ALSO, chúng ta cũng đưa ra cây truy vấn gốc ban đầu. Ngắn gọn, đầu ra từ hệ thống quy tắc là một danh sách 2 cây truy vấn tương ứng với các lệnh đó:

```
INSERT INTO shoelace_log VALUES (
        shoelace_data.sl_name, 6,
        current_user, current_timestamp )
  FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

Chúng được thực thị theo trật tự này, và điều đó là chính xác những gì quy tắc đó có nghĩa phải làm. Những thay thế và các định tính được thêm vào đảm bảo rằng, nếu truy vấn gốc ban đầu có thể là:

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

thì không khoản đầu vào lưu ký nào được viết. Trong trường hợp đó, cây truy vấn gốc ban đầu không chứa khoản đầu vào của danh sách đích đối với sl_avail, nên NEW.sl_avail sẽ được thay thế bằng shoelace_data.sl_avail. Vì thế, lệnh thêm đó được sinh ra bằng quy tắc đó là:

```
INSERT INTO shoelace_log VALUES (
        shoelace_data.sl_name, shoelace_data.sl_avail,
        current_user, current_timestamp )
  FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

và định tính đó sẽ không bao giờ là đúng.

Nó cũng làm việc nếu truy vấn gốc ban đầu sửa đổi nhiều hàng. Vì thế nếu ai đó đã đưa ra lệnh:

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

thì 4 hàng trong thực tế sẽ được cập nhật (sl1, sl2, sl3, và sl4). Nhưng sl3 có rồi sl_avail = 0. Trong trường hợp này, định tính các cây truy vấn gốc ban đầu là khác nhau và các kết quả đó trong cây truy vấn thêm:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
        current_user, current_timestamp
  FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
    AND shoelace_data.sl_color = 'black';
```

đang được quy tắc đó sinh ra. Cây truy vấn này chắc chắn sẽ chèn 3 khoản đầu vào lưu ký mới. Và điều đó tuyệt đối là đúng.

Ở đây chúng ta có thể thấy vì sao là quan trọng rằng cây truy vấn gốc ban đầu được thực thi cuối cùng. Nếu UPDATE đã được thực thi đầu tiên, thì tất cả các hàng có thể đã được thiết lập rồi về zero, vì thế việc lưu ký INSERT có thể không thấy bất kỳ hàng nào, trong đó 0 <> shoelace_data.sl_avail.

37.3.2. Cộng tác với các kiểu nhìn

Cách đơn giản để bảo vệ các quan hệ kiểu nhìn từ khả năng được nêu rằng ai đó có thể cố chạy INSERT, UPDATE, hoặc DELETE trong chúng là để các cây truy vấn đó được đưa đi. Vì thế chúng ta có thể tạo các quy tắc:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
        DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
        DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
        DO INSTEAD NOTHING;
```

Nếu ai đó bây giờ cố gắng làm bất kỳ hoạt động nào trong quan hệ kiểu nhìn shoe, thì hệ thống quy tắc sẽ áp dụng các quy tắc đó. Vì các quy tắc đó không có các hành động và là INSTEAD, nên danh sách kết quả của các cây truy vấn sẽ là rỗng và toàn bộ truy vấn đó sẽ trở thành không gì cả vì không có gì để tối ưu hóa hoặc thực thi sau khi hệ thống quy tắc được thực hiện xong với nó.

Cách phức tạp hơn để sử dụng hệ thống quy tắc là tạo các quy tắc mà viết lại cây truy vấn đó trong một quy tắc mà thực hiện hoạt động đúng trong các bảng thực tế. Để làm điều đó trong kiểu nhìn shoelace, chúng ta tạo các quy tắc sau:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
        DO INSTEAD
        INSERT INTO shoelace_data VALUES (
                NEW.sl_name,
                NEW.sl_avail,
                NEW.sl_color,
                NEW.sl_len,
                NEW.sl_unit
        );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
        DO INSTEAD
        UPDATE shoelace_data
             SET sl_name = NEW.sl_name,
                 sl_avail = NEW.sl_avail,
                 sl_color = NEW.sl_color,
                 sl_len = NEW.sl_len,
                 sl_unit = NEW.sl_unit
        WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
        DO INSTEAD
        DELETE FROM shoelace_data
         WHERE sl_name = OLD.sl_name;
```

Nếu bạn muốn hỗ trợ các truy vấn RETURNING trong kiểu nhìn đó, bạn cần làm cho các quy tắc đó đưa các mệnh đề RETURNING mà tính toán các hàng kiểu nhìn đó. Điều này thường khá tầm thường đối với các kiểu nhìn trong một bảng duy nhất, nhưng là khá nặng nhọc cho các kiểu nhìn liên kết như shoelace. Một ví dụ về trường hợp chèn là:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
        DO INSTEAD
        INSERT INTO shoelace_data VALUES (
                NEW.sl_name,
                NEW.sl_avail,
                NEW.sl_color,
                NEW.sl_len,
                NEW.sl_unit
        )
        RETURNING
                shoelace_data.*,
                (SELECT shoelace_data.sl_len * u.un_fact
                   FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

Lưu ý rằng quy tắc này hỗ trợ cả các truy vấn INSERT và INSERT RETURNING trong kiểu nhìn đó - mệnh đề RETURNING đơn giản sẽ bỏ qua đối với INSERT.

Bây giờ giả thiết là một lần nào đó, một gói dây giầy tới cửa hàng và một danh sách lớn các phụ kiện đi với nó. Nhưng bạn không muốn cập nhật bằng tay kiểu nhìn shoelace mỗi lần. Thay vào đó chúng ta thiết lập 2 bảng nhỏ: một bảng nơi mà chúng ta có thể chèn các khoản từ danh sách phụ kiện, và một bảng với một mẹo đặc biệt. Các lệnh tạo ra cho chúng là:

```
CREATE TABLE shoelace_arrive (
        arr_name text,
        arr_quant integer
);

CREATE TABLE shoelace_ok (
        ok_name text,
        ok_quant integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
        DO INSTEAD
        UPDATE shoelace
            SET sl_avail = sl_avail + NEW.ok_quant
         WHERE sl_name = NEW.ok_name;
```

Bây giờ bạn có thể điền bảng bằng các dữ liệu từ danh sách các phụ kiện:

```
SELECT * FROM shoelace_arrive;

arr_name | arr_quant
-------------+--------------
sl3         |          10
sl6         |          20
sl8         |          20
(3 rows)
```

Xem nhanh các dữ liệu hiện hành:

```
SELECT * FROM shoelace;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
------------+----------+----------+--------+----------+-------------
sl1         |        5 | black    |     80 | cm       |          80
sl2         |        6 | black    |    100 | cm       |         100
sl7         |        6 | brown    |     60 | cm       |          60
sl3         |        0 | black    |     35 | inch     |        88.9
sl4         |        8 | black    |     40 | inch     |      101.6
sl8         |        1 | brown    |     40 | inch     |      101.6
sl5         |        4 | brown    |      1 | m        |         100
sl6         |        0 | brown    |    0.9 | m        |          90
(8 rows)
```

Bây giờ hãy chuyển các dây giầy được đưa tới vào:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

và kiểm tra các kết quả:

```
SELECT * FROM shoelace ORDER BY sl_name;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
------------+----------+----------+--------+----------+-------------
sl1         |        5 | black    |     80 | cm       |          80
sl2         |        6 | black    |    100 | cm       |         100
sl7         |        6 | brown    |     60 | cm       |          60
```

```
sl4       |        8 | black   |    40 | inch    |    101.6
sl3       |       10 | black   |    35 | inch    |     88.9
sl8       |       21 | brown   |    40 | inch    |    101.6
sl5       |        4 | brown   |     1 | m       |      100
sl6       |       20 | brown   |   0.9 | m       |       90
(8 rows)

SELECT * FROM shoelace_log;

sl_name | sl_avail | log_who| log_when
------------+----------+-----------+--------------------------------------------------
sl7       |        6 | Al      | Tue Oct 20 19:14:45 1998 MET DST
sl3       |       10 | Al      | Tue Oct 20 19:25:16 1998 MET DST
sl6       |       20 | Al      | Tue Oct 20 19:25:16 1998 MET DST
sl8       |       21 | Al      | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

Đây là một con dường dài từ một INSERT ... SELECT tới các kết quả đó. Và mô tả của sự biến đổi cây truy vấn sẽ là cuối cùng trong chương này. Trước nhất, có đầu ra của trình phân tích cú pháp:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
        FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Bây giờ, quy tắc đầu tiên shoelace_ok_ins được áp dụng và biến điều này thành:

```
UPDATE shoelace
     SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
   FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
          shoelace_ok old, shoelace_ok new,
          shoelace shoelace
  WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

và đưa đi INSERT gốc ban đầu vào shoelace_ok. Truy vấn được viết lại này được truyền tới hệ thống quy tắc một lần nữa, và quy tắc shoelace_upd được áp dụng thứ 2 tạo ra:

```
UPDATE shoelace_data
    SET sl_name = shoelace.sl_name,
        sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
        sl_color = shoelace.sl_color,
        sl_len = shoelace.sl_len,
        sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
        shoelace_ok old, shoelace_ok new,
        shoelace shoelace, shoelace old,
        shoelace new, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Một lần nữa một quy tắc INSTEAD và cây truy vấn trước đó bị bỏ đi. Lưu ý là truy vấn này vẫn sử dụng kiểu nhìn shoelace. Nhưng hệ thống quy tắc còn chưa hoàn tất với bước này, nên nó tiếp tục và áp dụng quy tắc _RETURN lên nó, và chúng ta có:

```
UPDATE shoelace_data
    SET sl_name = s.sl_name,
        sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
        sl_color = s.sl_color,
        sl_len = s.sl_len,
        sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
```

```
        shoelace_ok old, shoelace_ok new,
        shoelace shoelace, shoelace old,
        shoelace new, shoelace_data shoelace_data,
        shoelace old, shoelace new,
        shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name;
```

Cuối cùng, quy tắc được áp dụng, tạo ra cây truy vấn thêm:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
        s.sl_avail + shoelace_arrive.arr_quant,
        current_user,
        current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
        shoelace_ok old, shoelace_ok new,
        shoelace shoelace, shoelace old,
        shoelace new, shoelace_data shoelace_data,
        shoelace old, shoelace new,
        shoelace_data s, unit u,
        shoelace_data old, shoelace_data new
        shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name
    AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

Sau đó hệ thống quy tắc chạy hết các quy tắc và trả về các cây truy vấn được sinh ra.

Vì thế chúng ta kết thúc với 2 cây truy vấn cuối cùng mà là tương đương với các lệnh SQL:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
        s.sl_avail + shoelace_arrive.arr_quant,
        current_user,
        current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
        shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name
    AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
     SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
   FROM shoelace_arrive shoelace_arrive,
        shoelace_data shoelace_data,
        shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
    AND shoelace_data.sl_name = s.sl_name;
```

Kết quả là các dữ liệu tới từ một quan hệ được chèn vào quan hệ khác, bị thay đổi thành các cập nhật trong quan hệ thứ 3, bị thay đổi thành việc cập nhật thứ 4 cộng với việc lưu ký mà cập nhật cuối cùng trong một quan hệ thứ 5 được giảm thành 2 truy vấn.

Có một ít chi tiết hơi xấu xí một chút. Nhìn vào 2 truy vấn đó, hóa ra là quan hệ `shoelace_data` xuất hiện 2 lần trong bảng dãy, nơi mà nó có thể chắc chắn được giảm thiểu thành 1 lần. Trình lên kế hoạch không xử trí nó và vì thế kế hoạch thực thi cho đầu ra các hệ thống quy tắc của `INSERT` sẽ là

```
Nested Loop
        -> Merge Join
                -> Seq Scan
                        -> Sort
                                -> Seq Scan on s
                -> Seq Scan
                        -> Sort
                                -> Seq Scan on shoelace_arrive
        -> Seq Scan on shoelace_data
```

trong khi việc bỏ qua khoản đầu vào bảng dãy thêm có thể gây ra một

```
Merge Join
        -> Seq Scan
                -> Sort
                        -> Seq Scan on s
        -> Seq Scan
                -> Sort
                        -> Seq Scan on shoelace_arrive
```

nó tạo ra chính xác các khoản đầu vào y hệt trong bảng lưu ký. Vì thế, hệ thống quy tắc đã gây ra sự quét dư thừa trong bảng shoelace_data mà tuyệt đối là không cần thiết. Và sự quét dư thừa y hệt được thực hiện một lần nữa trong UPDATE. Nhưng đây thực sự từng là một công việc nặng nhọc để làm cho tất cả điều đó thành có thể.

Bây giờ chúng ta thực hiện trình diễn cuối cùng về hệ thống quy tắc của PostgreSQL và sức mạnh của nó. Bạn hãy thêm vài đôi dây giầy với các màu thừa thãi vào cơ sở dữ liệu của bạn:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Chúng ta muốn làm một kiểu nhìn để kiểm tra các khoản đầu vào shoelace nào không phù hợp với bất kỳ màu của đôi giầy nào. Kiểu nhìn cho điều này là:

```
CREATE VIEW shoelace_mismatch AS
        SELECT * FROM shoelace WHERE NOT EXISTS
                (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Đầu ra của nó là:

```
SELECT * FROM shoelace_mismatch;

sl_name | sl_avail | sl_color   | sl_len | sl_unit | sl_len_cm
------------+----------+-------------+--------+----------+--------------
sl9         |        0 |      pink   |     35 | inch     |        88.9
sl10        |    1000 | magenta |     40 | inch     |      101.6
```

Bây giờ chúng ta muốn thiết lập nó sao cho các dây giầy khớp màu mà còn chưa nằm trong kho sẽ bị xóa khỏi cơ sở dữ liệu. Để làm điều này một chút khó cho PostgreSQL, chúng ta không xóa nó trực tiếp. Thay vào đó chúng ta tạo một kiểu nhìn nữa:

```
CREATE VIEW shoelace_can_delete AS
        SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

và thực hiện nó theo cách này:

```
DELETE FROM shoelace WHERE EXISTS
        (SELECT * FROM shoelace_can_delete
                WHERE sl_name = shoelace.sl_name);
```

Đây rồi:

```
SELECT * FROM shoelace;

sl_name | sl_avail | sl_color   | sl_len | sl_unit | sl_len_cm
------------+----------+------------+--------+---------+--------------
sl1         |        5 | black      |     80 | cm      |         80
sl2         |        6 | black      |    100 | cm      |        100
sl7         |        6 | brown      |     60 | cm      |         60
sl4         |        8 | black      |     40 | inch    |      101.6
sl3         |       10 | black      |     35 | inch    |       88.9
sl8         |       21 | brown      |     40 | inch    |      101.6
sl10        |     1000 | magenta    |     40 | inch    |      101.6
sl5         |        4 | brown      |      1 | m       |        100
sl6         |       20 | brown      |    0.9 | m       |         90
(9 rows)
```

Một DELETE trong một kiểu nhìn, với định tính của một truy vấn con mà tổng số sử dụng 4 kiểu nhìn lồng nhau/được liên kết, trong đó 1 trong số chúng có định tính của một truy vấn con gồm 1 kiểu nhìn và trong đó các cột kiểu nhìn được tính toán được sử dụng, biến thứ được viết lại thành một cây truy vấn duy nhất mà nó xóa các dữ liệu được yêu cầu từ một bảng thực.

Có thể chỉ có một ít tình huống như vậy trong thế giới thực, nơi mà một cấu trúc như vậy là cần thiết. Nhưng nó làm cho bạn cảm thấy thuận tiện là nó làm việc.

37.4. Quy tắc và quyền ưu tiên

Vì việc viết lại các truy vấn hệ thống quy tắc của PostgreSQL, các bảng/kiểu nhìn khác với các bảng/kiểu nhìn được sử dụng trong truy vấn gốc ban đầu là truy cập được. Khi các quy tắc cập nhật được sử dụng, thì điều này có thể bao gồm truy cập ghi tới các bảng.

Các quy tắc viết lại không có một chủ sở hữu riêng biệt. Chủ sở hữu của một quan hệ (bảng hoặc kiểu nhìn) tự động là chủ sở hữu của các quy tắc viết lại được xác định cho nó. Hệ thống quy tắc của PostgreSQL thay đổi hành vi của hệ thống kiểm soát truy cập mặc định. Các quan hệ được sử dụng vì các quy tắc được kiểm tra về các quyền ưu tiên của chủ sở hữu quy tắc, không phải việc triệu gọi quy tắc đó của người sử dụng. Điều này có nghĩa là một người sử dụng chỉ cần các quyền ưu tiên được yêu cầu cho các bảng/kiểu nhìn mà anh ta rõ ràng đặt tên trong các truy vấn của anh ta.

Ví dụ: một người sử dụng có một danh sách các số điện thoại trong đó vài số là riêng tư, các số khác là có ích cho tính bí mật của văn phòng. Anh ta có thể xây thứ sau đây:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
        SELECT person, CASE WHEN NOT private THEN phone END AS phone
        FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

Không ai, ngoại trừ anh ta (và các siêu người sử dụng cơ sở dữ liệu đó) có thể truy cập bảng phone_data table đó. Nhưng vì GRANT, tính bí mật có thể chạy một lệnh SELECT trong kiểu nhìn

phone_number. Hệ thống quy tắc sẽ viết lại SELECT từ phone_number thành SELECT from phone_data. Vì người sử dụng đó là chủ sở hữu của phone_number và vì thế là chủ sở hữu của quy tắc đó, truy cập đọc tới phone_data bây giờ được kiểm tra đối với các quyền ưu tiên của anh ta và truy vấn được cho phép. Sự kiểm tra việc truy cập phone_number cũng được thực hiện, nhưng điều này được làm đối với người sử dụng đang triệu gọi, vì thế không ai ngoài người sử dụng đó và sự bí mật có thể sử dụng nó.

Các quyền ưu tiên được kiểm tra theo từng quy tắc một. Vì thế tính bí mật tới bây giờ chỉ cho người có thể nhìn thấy các số điện thoại công khai. Nhưng tính bí mật có thể thiết lập một kiểu nhìn khác và trao sự truy cập tới kiểu nhìn đó cho công chúng. Sau đó, bất kỳ ai cũng có thể thấy các dữ liệu phone_number qua kiểu nhìn bí mật đó. Những gì sự bí mật đó không thể làm là tạo một kiểu nhìn mà trực tiếp truy cập được phone_data. (Thực sự anh ta có thể, nhưng nó sẽ không làm việc vì từng truy cập sẽ bị từ chối trong quá trình kiểm tra sự cho phép). Và ngay khi người sử dụng đó lưu ý, rằng sự bí mật được mở đối với kiểu nhìn phone_number của anh ta, thì anh ta có thể thu hồi sự truy cập của anh ta. Ngay lập tức, bất kỳ sự truy cập nào tới kiểu nhìn bí mật có thể sẽ thất bại.

Người ta có thể nghĩ rằng việc kiểm tra theo từng quy tắc này là một lỗ hổng về an toàn, nhưng trong thực tế không phải thế. Nhưng nếu nó không làm việc theo cách này, thì tính bí mật có thể thiết lập một bảng với các cột y hệt như phone_number và sao chép các dữ liệu tới đó một lần trong mỗi ngày. Rồi thì nó là các dữ liệu của riêng anh ta và anh ta có thể trao sự truy cập tới bất kỳ ai anh ta muốn. Một lệnh GRANT có nghĩa là, “Tôi tin bạn”. Nếu ai đó bạn tin làm điều ở trên, thì tới lúc nghĩ nó cần kết thúc và sau đó hãy sử dụng REVOKE.

Hãy lưu ý rằng trong khi các kiểu nhìn có thể được sử dụng để ẩn dấu các nội dung các cột nhất định bằng việc sử dụng kỹ thuật được chỉ ra ở trên, thì chúng không thể được sử dụng để giấu giếm một cách tin cậy các dữ liệu trong các hàng không được nhìn. Ví dụ, kiểu nhìn sau là không an toàn:

```
CREATE VIEW phone_number AS
        SELECT person, phone FROM phone_data WHERE phone NOT LIKE ’412%’;
```

Kiểu nhìn này có thể dường như an toàn, vì hệ thống quy tắc sẽ viết lại bất kỳ SELECT from phone_number nào thành SELECT from phone_data và thêm sự định tính mà chỉ các khoản đầu vào nơi mà phone không bắt đầu bằng 412 sẽ được cần tới. Nhưng nếu người sử dụng có thể tạo ra các hàm riêng của anh hoặc chị ta, thì không khó để thuyết phục trình lên kế hoạch để thực thi hàm do người sử dụng định nghĩa trước đối với biểu thức NOT LIKE.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
        RAISE NOTICE ’% => %’, $1, $2;
        RETURN true;
END
$$ LANGUAGE plpgsql COST 0.00000000000000000001;
SELECT * FROM phone_number WHERE tricky(person, phone);
```

Mỗi người và mỗi số điện thoại trong bảng phone_data sẽ được in như là một NOTICE, vì trình lên kế hoạch sẽ chọn thực thi hàm tricky không đắt giá trước NOT LIKE đắt giá hơn. Thậm chí nếu người sử dụng bị ngăn chặn khỏi việc định nghĩa các hàm mới, thì các hàm được xây dựng sẵn có thể được

sử dụng trong các cuộc tấn công tương tự. (Ví dụ, đưa ra các hàm bao gồm các đầu vào của chúng trong các thông điệp lỗi mà chúng sản xuất).

Những xem xét tương tự áp dụng để cập nhật các quy tắc. Trong các ví dụ của phần trước, chủ sở hữu của các bảng trong cơ sở dữ liệu ví dụ có thể trao các quyền ưu tiên SELECT, INSERT, UPDATE, và DELETE trong kiểu nhìn shoelace cho ai đó khác, nhưng chỉ SELECT trong shoelace_log. Hành động của quy tắc để ghi các khoản đầu vào lưu ký sẽ vẫn được thực thi thành công, và người sử dụng khác đó có thể thấy các khoản đầu vào lưu ký đó. Nhưng anh ta không thể tạo các khoản đầu vào giả mạo, anh ta cũng không thể điều khiển hoặc loại bỏ các khoản đang tồn tại. Trong trường hợp này, không có khả năng phá hoại các quy tắc bằng việc thuyết phục trình lên kế hoạch tùy biến trật tự các hoạt động, vì chỉ quy tắc mà tham chiếu tới shoelace_log là một INSERT không đủ điều kiện. Điều này có thể không đúng trong các kịch bản phức tạp hơn.

37.5. Quy tắc và tình trạng lệnh

Máy chủ PostgreSQL trả về một chuỗi tình trạng lệnh, như INSERT 149592 1, đối với mỗi lệnh nó nhận được. Điều này là đủ đơn giản khi không có các quy tắc liên quan, nhưng điều gì sẽ xảy ra khi truy vấn được các quy tắc viết ra?

Các quy tắc ảnh hưởng tới tình trạng lệnh như sau:

- Nếu không có quy tắc INSTEAD không điều kiện cho truy vấn, thì truy vấn được đưa ra ban đầu sẽ được thực thi, và tình trạng lệnh của nó được trả về như bình thường. (Nhưng lưu ý rằng nếu đã có bất ký quy tắc INSTEAD có điều kiện nào, thì sự phủ định các định tính của chúng sẽ được thêm vào truy vấn gốc ban đầu. Điều này có thể làm giảm số hàng nó xử lý, và nếu thế thì tình trạng được nêu sẽ bị ảnh hưởng).
- Nếu có bất kỳ quy tắc INSTEAD không có điều kiện nào đối với truy vấn đó, thì truy vấn gốc ban đầu sẽ không được thực thi hoàn toàn. Trong trường hợp này, máy chủ sẽ trả về tình trạng lệnh cho truy vấn cuối cùng mà đã được chèn vào bằng một quy tắc INSTEAD (có hoặc không có điều kiện) và ở dạng lệnh y hệt (INSERT, UPDATE, hoặc DELETE) như truy vấn gốc ban đầu. Nếu không truy vấn nào đáp ứng được các yêu cầu được bất kỳ quy tắc nào đó thêm vào, thì tình trạng lệnh được trả về chỉ ra dạng truy vấn gốc ban đầu và các số zero cho sự đếm hàng và các trường OID.

(Hệ thống này đã được thiết lập trong PostgreSQL 7.3. Trong các phiên bản trước đó, tình trạng lệnh có thể chỉ ra các kết quả khác nhau khi các quy tắc tồn tại).

Lập trình viên có thể đảm bảo rằng bất kỳ quy tắc INSTEAD được mong muốn nào cũng là một quy tắc mà thiết lập tình trạng lệnh trong trường hợp thứ 2, bằng việc trao cho nó tên quy tắc cuối cùng theo vần abc trong các quy tắc tích cực, sao cho nó được áp dụng cuối cùng.

37.6. Quy tắc so với trigger

Nhiều điều có thể được thực hiện bằng việc sử dụng các trigger cũng có thể được triển khai bằng việc sử dụng hệ thống quy tắc của PostgreSQL. Một trong những điều mà không thể được các quy tắc triển khai là vài dạng ràng buộc, đặc biệt là các khóa ngoại. Là có khả năng để đặt một quy tắc đủ điều kiện mà viết lại một lệnh về NOTHING nếu giá trị của một cột không xuất hiện trong một bảng khác. Nhưng sau đó dữ liệu đó âm thầm được đưa đi và đó không phải là một ý tưởng tốt. Nếu kiểm tra các dữ liệu hợp lệ được yêu cầu, và trong trường hợp có một giá trị không hợp lệ thì một thông điệp lỗi sẽ được sinh ra, nó phải được một trigger thực hiện.

Mặt khác, một trigger không thể được tạo ra trong các kiểu nhìn vì không có dữ liệu thực nào trong một quan hệ kiểu nhìn cả; tuy nhiên, các quy tắc INSERT, UPDATE, và DELETE có thể được tạo ra trong các kiểu nhìn.

Đối với những điều mà có thể được cả 2 thứ triển khai, thì nó tốt nhất phụ thuộc vào sự sử dụng của cơ sở dữ liệu. Một trigger được chạy cho bất kỳ hàng bị tác động nào một lần. Một quy tắc xử trí truy vấn hoặc sinh ra một truy vấn bổ sung. Vì thế nếu nhiều hàng bị ảnh hưởng trong một lệnh, thì một quy tắc đưa ra một lệnh dư thêm có khả năng là nhanh hơn so với một trigger được gọi cho từng hàng đơn nhất và phải thực thi các hoạt động của nó nhiều lần. Tuy nhiên, tiếp cận trigger về khái nhiệm là đơn giản hơn nhiều so với tiếp cận quy tắc, và là dễ dàng hơn nhiều cho những người mới làm quen để làm đúng.

Ở đây chúng ta chỉ ra một ví dụ về cách lựa chọn các quy tắc so với các trigger làm trong một tình huống. Có 2 bảng:

```
CREATE TABLE computer (
        hostname text, -- indexed
        manufacturer text -- indexed
);

CREATE TABLE software (
        software text, -- indexed
        hostname text – indexed
);
```

Cả 2 bảng đều có nhiều ngàn hàng và các chỉ số về hostname là duy nhất. Quy tắc hoặc trigger nên triển khai một ràng buộc xóa các hàng khỏi software mà tham chiếu tới một máy tính bị xóa.

Trigger có thể sử dụng lệnh này:

```
DELETE FROM software WHERE hostname = $1;
```

Vì trigger đó được gọi cho từng hàng riêng rẽ bị xóa khỏi computer, nó có thể chuẩn bị và lưu kế hoạch cho lệnh này và truyền giá trị hostname trong tham số đó. Quy tắc đó có thể được viết như:

```
CREATE RULE computer_del AS ON DELETE TO computer
        DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Bây giờ chúng ta xem xét các dạng xóa khác nhau. Trong trường hợp của một:

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

bảng computer được quét bằng chỉ số (nhanh), và lệnh được trigger đưa ra có thể cũng sử dụng một quét chỉ số (cũng nhanh). Lệnh dư thêm từ quy tắc đó có thể là:

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                        AND software.hostname = computer.hostname;
```

Vì có thiết lập các chỉ số phù hợp, trình lên kế hoạch sẽ tạo một kế hoạch của

```
Nestloop
        -> Index Scan using comp_hostidx on computer
        -> Index Scan using soft_hostidx on software
```

Vì thế có thể không khác nhiều về tốc độ giữa triển khai trigger và quy tắc.

Với sự xóa tiếp theo chúng ta muốn bỏ tất cả 2.000 máy tính, trong đó hostname bắt đầu với old. Có 2 lệnh có khả năng làm điều đó. Một là:

```
DELETE FROM computer WHERE hostname >= 'old'
                        AND hostname < 'ole'
```

Lệnh được quy tắc đó thêm vào sẽ là:

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                        AND software.hostname = computer.hostname;
```

với kế hoạch

```
Hash Join
        -> Seq Scan on software
        -> Hash
                -> Index Scan using comp_hostidx on computer
```

Lệnh có khả năng khác là:

```
DELETE FROM computer WHERE hostname ~ '^old';
```

nó làm cho kế hoạch thực thi sau đây đối với lệnh được quy tắc đó thêm vào:

```
Nestloop
        -> Index Scan using comp_hostidx on computer
        -> Index Scan using soft_hostidx on software
```

Điều này chỉ ra, rằng trình lên kế hoạch không thừa nhận rằng định tính cho hostname trong computer cũng có thể được sử dụng cho sự quét chỉ số trong software khi có nhiều biểu thức định tính được kết hợp với AND, là những gì nó làm trong phiên bản thông thường của lệnh. Trigger sẽ được triệu gọi một lần cho từng trong số 2.000 máy tính cũ mà phải bị xóa, và điều đó gây ra một sự quét chỉ số đối với computer và 2.000 sự quét chỉ số đối với software. Sự triển khai quy tắc sẽ làm điều này với 2 lệnh mà sử dụng các chỉ số. Và nó phụ thuộc vào toàn bộ kích thước của bảng software dù quy tắc đó sẽ vẫn là nhanh hơn trong tình huống quét tuần tự đó. 2.000 sự thực thi lệnh từ trigger đó đối với trình quản lý SPI mất chút thời gian, thậm chí nếu tất cả các khối chỉ số sẽ sớm nằm trong bộ nhớ tạm (cache).

Lệnh cuối cùng chúng ta xem là:

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Một lần nữa điều này làm cho nhiều hàng sẽ bị xóa khỏi computer. Vì thế trigger một lần nữa sẽ

chạy nhiều lệnh qua trình thực thi. Lệnh đó được quy tắc sinh ra sẽ là:

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

Kế hoạch cho lệnh đó một lần nữa sẽ là vòng lặp lồng nhau đối với 2 sự quét chỉ số, chỉ sử dụng một chỉ số khác trong computer:

```
Nestloop
        -> Index Scan using comp_manufidx on computer
        -> Index Scan using soft_hostidx on software
```

Trong bất kỳ trường hợp nào, các lệnh dư thêm từ hệ thống quy tắc sẽ ít nhiều độc lập với số các hàng bị ảnh hưởng trong một lệnh.

Tóm tắt là, các quy tắc sẽ chỉ là chậm hơn đáng kể so với các trigger nếu các hành động của chúng gây ra các liên kết lớn và chất lượng tồi tệ, một tình huống nơi mà trình lên kế hoạch thất bại.

Chương 38. Các ngôn ngữ thủ tục

PostgreSQL cho phép các hàm do người sử dụng định nghĩa được viết trong các ngôn ngữ khác ngoài SQL và C. Các ngôn ngữ đó thường được gọi là *các ngôn ngữ thủ tục* - PL (*Procedural Language*). Đối với một hàm được viết trong một ngôn ngữ thủ tục, máy chủ cơ sở dữ liệu không có tri thức được xây dựng sẵn về cách để diễn giải văn bản nguồn của hàm. Thay vào đó, nhiệm vụ này được tryền tới một trình điều khiển đặc biệt biết các chi tiết về ngôn ngữ đó. Trình điều khiển đó hoặc có thể làm tất cả công việc phân tích cú pháp, phân tích ngữ nghĩa, thực thi, ... cho bản thân nó, hoặc nó có thể phục vụ như là "manh mối" giữa PostgreSQL và triển khai đang tồn tại của ngôn ngữ lập trình. Bản thân trình điều khiển đó là một hàm ngôn ngữ C được biên dịch thành một đối tượng chia sẻ được và tải được theo yêu cầu, giống hệt bất kỳ hàm C nào khác.

Hiện hành, có 4 ngôn ngữ thủ tục sẵn có trong phân phối PostgreSQL tiêu chuẩn: PL/pgSQL (Chương 39), PL/Tcl (Chương 40), PL/Perl (Chương 41), và PL/Python (Chương 42). Có các ngôn ngữ thủ tục bổ sung sẵn sàng mà không được đưa vào trong phân phối lõi. Phụ lục G có thông tin về việc tìm chúng. Trong các ngôn ngữ bổ sung khác có thể được những người sử dụng định nghĩa; cơ sở của việc phát triển một ngôn ngữ thủ tục được đề cập tới trong Chương 49.

38.1. Cài đặt các ngôn ngữ thủ tục

Ngôn ngữ thủ tục phải được "cài đặt" trong từng cơ sở dữ liệu trong đó nó sẽ được sử dụng. Nhưng các ngôn ngữ thủ tục được cài đặt trong cơ sở dữ liệu `template1` là tự động có sẵn trong tất cả các cơ sở dữ liệu được tạo ra sau đó, vì các khoản đầu vào của chúng trong `template1` sẽ được sao chép bằng lệnh `CREATE DATABASE`. Vì thế người quản trị cơ sở dữ liệu có thể quyết định các ngôn ngữ nào sẵn sàng trong cơ sở dữ liệu nào và có thể làm cho vài ngôn ngữ sẵn sàng mặc định nếu anh ta chọn.

Đối với các ngôn ngữ được cung cấp với phân phối tiêu chuẩn, chỉ cần thiết để thực thi `CREATE LANGUAGE language_name` để cài đặt ngôn ngữ đó vào cơ sở dữ liệu hiện hành.

Như một sự lựa chọn, chương trình `createlang` có thể được sử dụng để thực hiện điều này từ dòng lệnh của trình biên dịch shell. Ví dụ, để cài đặt ngôn ngữ PL/Perl vào cơ sở dữ liệu `template1`, hãy sử dụng:

```
createlang plperl template1
```

Thủ tục chỉ dẫn được mô tả bên dưới chỉ được khuyến cáo cho việc cài đặt các ngôn ngữ tùy biến mà `CREATE LANGUAGE` không biết.

Cài đặt Ngôn ngữ Thủ tục Chỉ dẫn (Manual Procedural Language Installation)

Một ngôn ngữ thủ tục được cài đặt trong một cơ sở dữ liệu theo 5 bước, chúng phải được triển khai bằng một siêu người sử dụng cơ sở dữ liệu. (Đối với các ngôn ngữ được biết để `CREATE LANGUAGE`, các bước thứ 2 tới 4 có thể được bỏ qua, vì chúng sẽ được triển khai tự động nếu cần).

1. Đối tượng được chia sẻ cho trình điều khiển ngôn ngữ phải được biên dịch và được cài đặt vào một thư mục thư viện phù hợp. Điều này làm việc theo cách y hệt như việc xây dựng và

cài đặt các module với các hàm C thông thường do người sử dụng định nghĩa làm; xem Phần 35.9.6. Thông thường, trình điều khiển ngôn ngữ sẽ phụ thuộc vào thư viện bên ngoài cung cấp máy ngôn ngữ lập trình thực tế; nếu thế, thì điều đó cũng phải được cài đặt.

2. Trình điều khiển phải được khai báo với lệnh

```
CREATE FUNCTION handler_function_name()
        RETURNS language_handler
        AS 'path-to-shared-object'
        LANGUAGE C;
```

Dạng trả về đặc biệt của language_handler nói cho hệ thống cơ sở dữ liệu rằng hàm này không trả về một trong những dạng dữ liệu SQL được định nghĩa và không sử dụng được trực tiếp trong các lệnh SQL.

3. Như một sự lựa chọn, trình điều khiển ngôn ngữ có thể đưa ra một hàm của trình điều khiển "tại chỗ" (inline) mà thực thi các khối mã nặc danh (các lệnh DO) được viết trong ngôn ngữ này. Nếu một hàm trình điều khiển tại chỗ được ngôn ngữ đó cung cấp, hãy khai báo nó với một lệnh như:

```
CREATE FUNCTION inline_function_name(internal)
        RETURNS void
        AS 'path-to-shared-object'
        LANGUAGE C;
```

4. Như một sự lựa chọn, trình điều khiển ngôn ngữ đó có thể đưa ra một hàm "của trình kiểm tra hợp lệ" (validator) mà kiểm tra một định nghĩa hàm xem có đúng hay không mà không cần thực sự thực thi nó. Hàm của trình kiểm tra hợp lệ được CREATE FUNCTION (nếu nó tồn tại) gọi. Nếu một hàm của trình kiểm tra hợp lệ được ngôn ngữ đó cung cấp, hãy khai báo nó với một lệnh như

```
CREATE FUNCTION validator_function_name(oid)
        RETURNS void
        AS 'path-to-shared-object'
        LANGUAGE C;
```

5. PL phải được khai báo với lệnh

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
        HANDLER handler_function_name
        [INLINE inline_function_name]
        [VALIDATOR validator_function_name] ;
```

Từ khóa tùy chọn TRUSTED chỉ định rằng ngôn ngữ đó không trao sự truy cập tới dữ liệu mà nếu khác thì người sử dụng có thể không có. Các ngôn ngữ tin cậy được thiết kế cho những người sử dụng thông thường cơ sở dữ liệu (những người không có quyền ưu tiên của siêu người sử dụng) và cho phép họ an toàn tạo các hàm và thủ tục trigger. Vì các hàm PL được thực thi bên trong máy chủ cơ sở dữ liệu, cờ TRUSTED chỉ nên được trao cho các ngôn ngữ mà không cho phép truy cập bên trong các máy chủ cơ sở dữ liệu hoặc hệ thống tệp. Các ngôn ngữ PL/pgSQL, PL/Tcl, và PL/Perl được coi là tin cậy; các ngôn ngữ PL/TclU, PL/PerlU, và PL/PythonU được thiết kế để cung cấp chức năng không giới hạn và sẽ không

được đánh dấu là tin cậy.

Ví dụ 38-1 chỉ cách thủ tục cài đặt chỉ dẫn có thể làm việc với ngôn ngữ PL/Perl.

Ví dụ 38-1. Cài đặt chỉ dẫn của PL/Perl

Lệnh sau đây nói cho máy chủ cơ sở dữ liệu nơi để tìm đối tượng được chia sẻ cho hàm của trình điều khiển lời gọi của ngôn ngữ PL/Perl:

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
        '$libdir/plperl' LANGUAGE C;
```

PL/Perl có một hàm của trình điều khiển tại chỗ và một hàm của trình kiểm tra hợp lệ, nên chúng ta cũng khai báo cho chúng:

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
        '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
        '$libdir/plperl' LANGUAGE C;
```

Lệnh:

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
        HANDLER plperl_call_handler
        INLINE plperl_inline_handler
        VALIDATOR plperl_validator;
```

sau đó xác định rằng các hàm được khai báo trước đó sẽ không được triệu gọi cho các hàm và các thủ tục trigger nơi mà thuộc tính ngôn ngữ đó là plperl.

Trong một cài đặt PostgreSQL mặc định, trình điều khiển cho ngôn ngữ PL/pgSQL được xây dựng và được cài đặt vào thư mục "thư viện"; xa hơn, bản thân ngôn ngữ PL/pgSQL được cài đặt trong tất cả các cơ sở dữ liệu. Nếu sự hỗ trợ Tcl được thiết lập cấu hình, thì các trình điều khiển cho PL/Tcl và PL/TclU sẽ được xây dựng và được cài đặt trong thư mục thư viện đó, nhưng bản thân ngôn ngữ đó không được cài đặt trong bất kỳ cơ sở dữ liệu nào một cách mặc định. Tương tự, các trình điều khiển PL/Perl và PL/PerlU được xây dựng và được cài đặt nếu sự hỗ trợ Perl được thiết lập cấu hình, và trình điều khiển PL/PythonU được cài đặt nếu sự hỗ trợ Python được thiết lập cấu hình, nhưng các ngôn ngữ đó không được cài đặt mặc định.

Chương 39. PL/pgSQL - Thủ tục SQL

39.1. Tổng quan

PL/pgSQL là một ngôn ngữ thủ tục tải được cho hệ thống cơ sở dữ liệu PostgreSQL. Các mục tiêu thiết kế của PL/pgSQL vốn để tạo một ngôn ngữ thủ tục tải được mà.

- có thể được sử dụng để tạo các hàm và các thủ tục trigger,
- thêm các cấu trúc kiểm soát cho ngôn ngữ SQL,
- có thể thực hiện các tính toán phức tạp,
- kế thừa tất cả các dạng, các hàm và các toán tử do người sử dụng định nghĩa,
- có thể được máy chủ xác định là tin cậy được,
- dễ dàng sử dụng

Các hàm được tạo ra với PL/pgSQL có thể được sử dụng ở bất kỳ nơi đâu mà các hàm được xây dựng sẵn có thể được sử dụng. Ví dụ, có khả năng tạo các hàm tính toán điều kiện phức tạp và sau đó sử dụng chúng để định nghĩa các toán tử hoặc sử dụng chúng trong các biểu thức chỉ số.

Trong PostgreSQL 9.0 và sau này, PL/pgSQL được cài đặt mặc định. Tuy nhiên nó vẫn là module có thể tải được, nên đặc biệt những người quản trị có ý thức về an toàn có thể chọn loại bỏ nó.

39.1.1. Ưu điểm của việc sử dụng PL/gpSQL

SQL là ngôn ngữ mà PostgreSQL và hầu hết các cơ sở dữ liệu quan hệ sử dụng như là ngôn ngữ truy vấn. Nó là khả chuyển và dễ học. Nhưng mỗi lệnh SQL phải được máy chủ cơ sở dữ liệu thực thi một cách riêng rẽ.

Điều đó có nghĩa là ứng dụng máy trạm của bạn phải gửi từng truy vấn tới máy chủ cơ sở dữ liệu, chờ nó được xử lý, nhận và xử lý các kết quả, tiến hành vài tính toán, sau đó gửi tiếp các truy vấn tới máy chủ. Tất cả điều này chịu truyền thông liên các tiến trình và sẽ luôn chịu tổng chi phí mạng nếu máy trạm của bạn là trên một máy tính khác với máy chủ cơ sở dữ liệu.

Với PL/pgSQL thì bạn có thể nhóm một khối tính toán và một loạt các truy vấn bên trong máy chủ cơ sở dữ liệu, vì thế có sức mạnh của một ngôn ngữ thủ tục và dễ dàng sử dụng SQL, nhưng với sự tiết kiệm đáng kể tổng chi phí truyền thông của máy trạm/máy chủ.

- Các chuyến đi vòng thêm giữa máy trạm và máy chủ sẽ được loại trừ
- Các kết quả trung gian mà máy trạm không cần sẽ không phải bị sắp đặt hoặc truyền giữa máy chủ và máy trạm
- Nhiều vòng phân tích cú pháp truy vấn có thể tránh được

Điều này có thể gây ra sự gia tăng hiệu năng đáng kể khi được so sánh với một ứng dụng không sử dụng các hàm được lưu trữ.

Hơn nữa, với PL/pgSQL thì bạn có thể sử dụng tất cả các dạng dữ liệu, các toán tử và các hàm SQL.

39.1.2. Đối số và các dạng dữ liệu kết quả được hỗ trợ

Các hàm được viết trong PL/pgSQL có thể chấp nhận như các đối số mà bất kỳ dạng dữ liệu vô hướng hoặc mảng nào cũng được máy chủ hỗ trợ, và chúng có thể trả về một kết quả của bất kỳ dạng nào đó. Chúng cũng có thể chấp nhận hoặc trả về bất kỳ dạng tính tổng nào (dạng hàng) được tên chỉ định. Cũng có khả năng để khai báo một hàm PL/pgSQL như là record trả về, điều có nghĩa là kết quả đó là một dạng hàng mà các cột của nó được đặc tả trong truy vấn gọi xác định, như được thảo luận trong Phần 7.2.1.4.

Các hàm PL/pgSQL có thể được khai báo để chấp nhận một số các đối số biến bằng việc sử dụng trình đánh dấu VARIADIC. Điều này làm việc theo cách thức chính xác y hệt như đối với các hàm SQL, như được thảo luận trong Phần 35.4.5.

Các hàm PL/pgSQL cũng có thể được khai báo để chấp nhận và trả về các dạng đa hình anyelement, anyarray, anynonarray, và anyenum. Các dạng dữ liệu thực sự được một hàm đa hình điều khiển có thể khác nhau từ lời gọi này tới lời gọi kia, như được thảo luận trong Phần 35.2.5. Một ví dụ được chỉ ra trong Phần 39.3.1.

Các hàm PL/pgSQL cũng có thể được khai báo để trả về một "tập hợp" (hoặc bảng) của bất kỳ dạng dữ liệu nào mà có thể được trả về như một trường hợp cá biệt đơn nhất. Một hàm như vậy sinh ra đầu ra của nó bằng việc thực thi RETURN NEXT cho từng phần tử có mong muốn của tập kết quả đó, hoặc bằng việc sử dụng RETURN QUERY cho đầu ra kết quả của việc đánh giá một truy vấn.

Cuối cùng, một hàm PL/pgSQL có thể được khai báo để trả về void nếu nó không có giá trị trả về hữu dụng.

Các hàm PL/pgSQL cũng có thể được khai báo với các tham số đầu ra tại chỗ của một đặc tả rõ ràng của dạng trả về. Điều này không thêm bất kỳ khả năng cơ bản nào cho ngôn ngữ đó, nhưng nó thường là thuận tiện, đặc biệt cho việc trả về nhiều giá trị. Dấu hiệu RETURNS TABLE cũng có thể được sử dụng tại chỗ của RETURNS SETOF.

Các ví dụ đặc biệt xuất hiện trong Phần 39.3.1 và Phần 39.6.1.

39.2. Cấu trúc của PL/pgSQL

PL/pgSQL là một ngôn ngữ khối có cấu trúc. Văn bản phức tạp của một định nghĩa hàm phải là một khối. Một khối được định nghĩa như là:

```
[ <<label>> ]
[ DECLARE
        declarations ]
BEGIN
        statements
END [ label ];
```

Từng khai báo và từng lệnh trong một khối kết thúc bằng dấu chấm phẩy. Một khối xuất hiện trong một khối khác phải có một dấu chấm phẩy sau END, như được chỉ ra ở trên; tuy nhiên END cuối cùng kết thúc thân hàm không yêu cầu một dấu chấm phẩy.

> **Mẹo**: Một sai lầm phổ biến là viết một dấu chấm phẩy ngay sau BEGIN. Điều này là sai và sẽ gây ra một lỗi cú pháp.

Một *nhãn (label)* chỉ cần thiết nếu bạn muốn nhận diện khối đó để sử dụng trong một lệnh EXIT, hoặc để định tính tên các biến được khai báo trong khối đó. Nếu một nhãn được đưa ra sau END, thì nó phải khớp với nhãn ở đầu của khối đó.

Tất cả các từ khóa là phân biệt chữ hoa chữ thường. Các mã định danh được chuyển đổi ngầm thành các chữ thường trừ phi có các dấu trích dẫn, hệt như chúng là trong các lệnh SQL thông thường.

Các chú giải làm việc theo cách y hệt trong mã PL/pgSQL như trong SQL thông thường. Một đúp bản dấu gạch ngang (--) bắt đầu một chú giải mà mở rộng tới cuối của dòng. Một /* bắt đầu một khối chú giải mà mở rộng tới nơi có */. Khối chú giải lồng nhau.

Bất kỳ lệnh nào trong phần lệnh của một khối cũng có thể là một *khối con (subblock)*. Các khối con có thể được sử dụng cho việc nhóm lại một cách logic hoặc bản địa hóa các biến cho một nhóm nhỏ các lệnh. Các biến được khai báo trong một khối con đeo mặt nạ cho bất kỳ biến được đặt tên tương tự nào của các khối bên ngoài cho khoảng thời gian của khối con đó; nhưng bạn có thể truy cập các biến bên ngoài bất kỳ cách gì nếu bạn định tính các tên của chúng với nhãn khối của chúng. Ví dụ:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
        quantity integer := 30;
BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
        quantity := 50;
        --
        -- Create a subblock
        --
        DECLARE
                quantity integer := 80;
        BEGIN
                RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
                RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50
        END;

        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50

        RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **Lưu ý**: Thực sự có một "khối bên ngoài" ẩn xung quanh thân của bất kỳ hàm PL/pgSQL nào. Khối này cung cấp các khai báo các tham số hàm (nếu có), cũng như vài biến đặc biệt như FOUND (xem Phần 39.5.5). Khối bên ngoài được gắn nhãn với tên hàm đó, nghĩa là các tham số và các biến đặc biệt đó có thể được định tính với tên hàm đó.

Là quan trọng để không lẫn lộn sử dụng BEGIN /END cho việc nhóm các lệnh trong PL/pgSQL với các lệnh SQL có tên tương tự để kiểm soát giao dịch. BEGIN /END của PL/pgSQL chỉ cho việc gộp nhóm; chúng không bắt đầu hoặc kết thúc một giao dịch. Các hàm và các thủ tục trigger luôn được thực thi trong một giao dịch được một truy vấn bên ngoài thiết lập - chúng không thể bắt đầu hoặc

đệ trình giao dịch đó, vì có thể không có ngữ cảnh cho chúng để thực thi. Tuy nhiên một khối có chứa một mệnh đề EXCEPTION sẽ hình thành có hiệu quả một giao dịch con có thể quay ngược lại mà không hề gây ảnh hưởng tới giao dịch bên ngoài đó. Để biết thêm, xem Phần 39.6.5.

39.3. Khai báo

Tất cả các biến được sử dụng trong một khối phải được khai báo trong phần các khai báo của khối đó. (Ngoại trừ duy nhất là biến lặp của một vòng lặp FOR nhắc đi nhắc lại qua một dãy các giá trị số nguyên là tự động được khai báo như một biến số nguyên, và tương tự biến lặp của vòng lặp FOR nhắc đi nhắc lại qua kết quả của con trỏ được khai báo tự động như một biến bản ghi).

Các biến PL/pgSQL có thể có bất kỳ dạng dữ liệu SQL nào, như integer, varchar, và char.

Đây là vài ví dụ về các khai báo biến:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Cú pháp chung của khai báo biến là:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

Mệnh đề DEFAULT, nếu được đưa ra, chỉ định giá trị ban đầu được phân công cho biến đó khi khối được đưa vào. Nếu mệnh đề DEFAULT không được đưa ra thì biến đó được khởi tạo cho SQL giá trị null. Lựa chọn CONSTANT ngăn chặn biến khỏi bị chỉ định, sao cho giá trị của nó sẽ giữ là hằng số trong thời gian của khối đó. Nếu NOT NULL được chỉ định, thì một sự chỉ định giá trị null sẽ gây ra một lỗi thời gian chạy. Tất cả các biến được khai báo như là NOT NULL phải có một giá trị mặc định nonnull (không null) được chỉ định.

Giá trị mặc định của biến được đánh giá và được chỉ định cho biến đó mỗi lần khối được đưa vào (không chỉ một lần cho lời gọi hàm). Vì thế, ví dụ, việc chỉ định now() cho một biến có dạng timestamp làm cho biến đó sẽ có thời gian của lời gọi hàm hiện hành, chứ không phải thời gian khi hàm đó đã được biên dịch sẵn. Các ví dụ:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

39.3.1. Khai báo tham số hàm

Các tham số được truyền tới các hàm được đặt tên với các mã định danh $1, $2, ... Như một sự lựa chọn, các tên hiệu (alias) có thể được khai báo cho các tên tham số $n vì khả năng đọc được sẽ được gia tăng. Hoặc các tên hiệu hoặc mã nhận diện số sau đó có thể được sử dụng để tham chiếu tới giá trị tham số.

Có 2 cách để tạo một tên hiệu. Cách được ưu tiên là trao một tên cho tham số đó trong lệnh CREATE

FUNCTION, ví dụ:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
        RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Cách khác, là cách duy nhất có sẵn trước PostgreSQL 8.0, là khai báo rõ ràng một tên hiệu, sử dụng cú pháp khai báo

name ALIAS FOR $n;

Ví dụ y hệt trong dạng này trông giống như:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
        subtotal ALIAS FOR $1;
BEGIN
        RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Lưu ý**: 2 ví dụ đó là không tương đương nhau. Trong trường hợp đầu, subtotal có thể được tham chiếu như là referenced as, nhưng trong trường hợp thứ 2 thì có thể không. (Chúng ta đã gắn một nhãn vào khối bên trong, thay vào đó, subtotal có thể được định tính với nhãn đó).

Vài ví dụ nữa:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
        v_string ALIAS FOR $1;
        index ALIAS FOR $2;
BEGIN
        -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
        RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Khi một hàm PL/pgSQL được khai báo với các tham số đầu ra, thì các tham số đầu ra sẽ được trao các tên $n và các tên hiệu tùy chọn theo đúng cách y hệt như các tham số đầu vào thông thường. Tham số đầu ra là một biến có hiệu quả mà bắt đầu bằng NULL; nó sẽ được chỉ định trong quá trình thực thi hàm. Giá trị cuối cùng của tham số đó là những gì được trả về. Ví dụ, các ví dụ về thuế bán hàng cũng có thể được thực hiện theo cách này.

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
        tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Lưu ý rằng chúng ta đã bỏ qua RETURNS real - chúng ta có thể bao gồm nó, nhưng nó có thể là dư thừa. Các tham số đầu ra là hữu dụng nhất khi trả về nhiều giá trị. Một ví dụ thông thường là:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
        sum := x + y;
        prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Như được thảo luận trong Phần 35.4.4, hiệu quả này tạo ra một dạng bản ghi nặc danh cho các kết quả của hàm. Nếu mệnh đề RETURNS được đưa ra, thì nó phải nói RETURNS record.

Một cách khác để khai báo hàm PL/pgSQL là với RETURNS TABLE, ví dụ:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
        RETURN QUERY SELECT quantity, quantity * price FROM sales
                        WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Điều này chính xác tương đương với việc khai báo một hoặc nhiều tham số OUT hơn và chỉ định RETURNS SETOF *sometype*.

Khi dạng trả về của một hàm PL/pgSQL được khai báo như một dạng đa hình (anyelement, anyarray, anynonarray, hoặc anyenum), thì một tham số đặc biệt $0 sẽ được tạo ra. Dạng dữ liệu của nó là dạng trả về thực của hàm, như được suy diễn ra từ các dạng đầu vào thực (xem Phần 35.2.5). Điều này cho phép hàm truy cập dạng trả về thực như được chỉ ra trong Phần 39.3.3. $0 được khởi tạo ở null và có thể được hàm đó sửa đổi, nên nó có thể được sử dụng để giữ giá trị trả về nếu mong muốn, dù điều đó không được yêu cầu. $0 cũng có thể được đưa ra như một tên hiệu. Ví dụ, hàm này làm việc trong bất kỳ dạng dữ liệu nào mà có một toán tử +:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
        result ALIAS FOR $0;
BEGIN
        result := v1 + v2 + v3;
        RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Hiệu ứng y hệt có thể có được bằng việc khai báo một hoặc nhiều tham số đầu ra hơn như là các dạng đa hình. Trong trường hợp này thì tham số đặc biệt $0 không được sử dụng; bản thân các tham số đầu ra đó phục vụ cho mục đích y hệt. Ví dụ:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
        sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

39.3.2. Tên hiệu (ALIAS)

newname ALIAS FOR oldname;

Cú pháp ALIAS là chung hơn so với được gợi ý trong phần trước: bạn có thể khai báo một tên hiệu cho bất kỳ biến nào, không chỉ các tham số hàm. Sử dụng thực tế chính cho điều này là để chỉ định một tên khác cho các biến với các tên được định nghĩa sẵn, như NEW hoặc OLD trong một thủ tục trigger. Các ví dụ:

```
DECLARE
        prior ALIAS FOR old;
        updated ALIAS FOR new;
```

Vì ALIAS tạo ra 2 cách khác nhau để đặt tên cho cùng một đối tượng, nên sử dụng không bị hạn chế có thể là gây lúng túng. Là tốt nhất để sử dụng nó chỉ cho mục đích ghi đè tên được xác định sẵn.

39.3.3. Sao chép dạng

variable%TYPE

%TYPE đưa ra dạng dữ liệu của một biến hoặc cột của bảng. Bạn có thể sử dụng điều này để khai báo các biến mà sẽ giữ các giá trị của cơ sở dữ liệu. Ví dụ, giả sử bạn có một cột có tên là user_id trong bảng users của bạn. Để khai báo một biến với dạng dữ liệu y hệt như users.user, bạn viết:

```
user_id users.user_id%TYPE;
```

Bằng việc sử dụng %TYPE, bạn không cần biết dạng dữ liệu của cấu trúc mà bạn đang tham chiếu, và quan trọng nhất, nếu dạng dữ liệu của khoản được tham chiếu thay đổi trong tương lai (ví dụ: bạn thay đổi dạng của user_id từ integer thành real), thì bạn có thể không cần thay đổi định nghĩa hàm của bạn.

%TYPE đặc biệt có giá trị trong các hàm đa hình, vì các dạng dữ liệu đó cần thiết cho các biến nội bộ có thể thay đổi từ lời gọi này sang lời gọi tiếp theo. Các biến đúng phù hợp có thể được tạo ra bằng việc áp dụng %TYPE cho các đối số hàm hoặc các chỗ trống để sẵn của kết quả.

39.3.4. Dạng hàng

name table_name%ROWTYPE;
name composite_type_name;

Biến của dạng tính tổng được gọi là một biến *hàng* (hoặc biến *dạng hàng*) Một biến như vậy có thể giữ toàn bộ hàng kết quả truy vấn của một SELECT hoặc FOR, miễn là tập hợp cột của truy vấn đó khớp với dạng biến được khai báo. Các trường riêng rẽ của giá trị hàng được truy cập bằng việc sử dụng dấu chấm thông thường, ví dụ rowvar.field.

Biến hàng có thể được khai báo để có dạng y hệt như các hàng của một bảng hoặc kiểu nhìn đang tồn tại, bằng việc sử dụng dấu hiệu table_name%ROWTYPE; hoặc nó có thể được khai báo bằng việc trao một tên dạng tính tổng. (Vì mỗi bảng có một dạng tính tổng có liên quan của tên y hệt, thực sự không là vấn đề trong PostgreSQL liệu bạn có viết hay không viết %ROWTYPE. Nhưng dạng %ROWTYPE là khả chuyển hơn).

Các tham số cho một hàm có thể là các dạng tính tổng (các hàng bảng hoàn chỉnh). Trong trường

hợp đó, mã định danh tương ứng $n sẽ là một biến hàng, và các trường có thể được chèn từ nó, ví dụ $1.user_id. Chỉ các cột do người sử dụng định nghĩa của một hàng của bảng sẽ truy cập được trong một biến dạng hàng, chứ không phải OID hoặc các cột hệ thống khác (vì hàng đó có thể là từ một kiểu nhìn). Các trường của dạng hàng kế thừa kích cỡ hoặc độ chính xác trường của bảng đối với các dạng dữ liệu như char(n).

Đây là một ví dụ về sử dụng các dạng tính tổng. table1 và table2 là các bảng đang tồn tại có ít nhất các trường được nhắc tới:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
        t2_row table2%ROWTYPE;
BEGIN
        SELECT * INTO t2_row FROM table2 WHERE ... ;
        RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

39.3.5. Dạng bản ghi

```
name RECORD;
```

Các biến bản ghi là tương tự với các biến dạng hàng, nhưng chúng không có cấu trúc được xác định sẵn. Chúng lấy cấu trúc hàng thực tế của hàng mà chúng được chỉ định trong một lệnh SELECT hoặc FOR. Cấu trúc con của một biến bản ghi có thể thay đổi mỗi lần nó được chỉ định. Hệ quả của điều này là cho tới khi một biến bản ghi được chỉ định lần đầu, nó không có cấu trúc con, và bất kỳ sự cố gắng nào để truy cập một trường trong nó sẽ kéo theo một lỗi thời gian chạy.

Lưu ý rằng RECORD không là dạng dữ liệu đúng, chỉ là một nơi giữ chỗ. Người ta cũng nên nhận thức được rằng khi một hàm PL/pgSQL được khai báo để trả về dạng record, thì điều này không phải là khái niệm y hệt như một biến bản ghi, thậm chí dù một hàm như vậy có lẽ sử dụng một biến bản ghi để giữ kết quả của nó. Trong cả 2 trường hợp cấu trúc hàng thực sự là không rõ khi mà hàm đó được viết, nhưng đối với một hàm trả về record cấu trúc hàng thực được xác định khi việc gọi truy vấn được phân tích cú pháp, trong khi một biến bản ghi có thể thay đổi cấu trúc hàng của nó ngay tại chỗ.

39.4. Biểu thức

Tất cả các biểu thức được sử dụng trong các lệnh PL/pgSQL được xử lý bằng việc sử dụng trình thực thi SQL chính của máy chủ. Ví dụ, khi bạn viết một lệnh PL/pgSQL như

```
IF expression THEN ...
```

PL/pgSQL sẽ đánh giá biểu thức đó bằng việc nuôi một truy vấn như

```
SELECT expression
```

cho máy SQL chính. Trong khi thực hiện lệnh SELECT, bất kỳ trường hợp nào các tên biến của

PL/pgSQL cũng được các tham số thay thế, như được thảo luận chi tiết trong Phần 39.10.1. Điều này cho phép kế hoạch truy vấn cho SELECT đó sẽ được chuẩn bị chỉ một lần và sau đó được sử dụng lại cho các đánh giá sau này với các giá trị các biến khác nhau. Vì thế, những gì thực sự xảy ra trong lần sử dụng đầu tiên một biểu thức, về cơ bản, là một lệnh PREPARE. Ví dụ, nếu chúng ta đã khai báo 2 biến số nguyên x và y, và chúng ta viết

```
IF x < y THEN ...
```

thì điều gì đằng sau các kịch bản đó là tương đương với

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

và sau đó lệnh được chuẩn bị sẵn này là EXECUTE d cho từng sự thực thi của lệnh IF, với các giá trị hiện hành của các biến PL/pgSQL được cung cấp như các giá trị tham số. Kế hoạch truy vấn đó được chuẩn bị theo cách này được lưu suốt đời một kết nối cơ sở dữ liệu, như được mô tả trong Phần 39.10.2. Thường thì các chi tiết đó sẽ không là quan trọng đối với một người sử dụng PL/gpSQL, nhưng chúng là hữu dụng để biết khi cố gắng chuẩn đoán một vấn đề.

39.5. Lệnh cơ bản

Trong phần này và các phần tiếp sau, chúng ta mô tả tất cả các dạng lệnh sẽ được PL/pgSQL hiểu rõ ràng. Bất kỳ điều gì không được thừa nhận như một trong các dạng lệnh đó được giả thiết sẽ là một lệnh SQL và được gửi tới máy cơ sở dữ liệu chính để thực thi, như được mô tả trong Phần 39.5.2 và Phần 39.5.3.

39.5.1. Chỉ định

Chỉ định một giá trị cho một biến PL/pgSQL được viết như:

```
variable := expression;
```

Như được giải thích trước đó, biểu thức trong một lệnh như vậy được đánh giá bằng phương tiện của một lệnh SQL SELECT được gửi tới máy cơ sở dữ liệu chính. Biểu thức đó phải lấy một giá trị duy nhất (có khả năng là một giá trị hàng, nếu biến đó là một hàng hoặc biến bản ghi). Biến đích có thể là một biến đơn giản (được định tính tùy chọn với một tên khối), một trường của một biến bản ghi hoặc hàng, hoặc một phần tử của một mảng mà là một biến đơn giản hoặc trường.

Nếu dạng dữ liệu kết quả của biểu thức đó không phù hợp với dạng dữ liệu của biến đó, hoặc biến có một kích cỡ/độ chính xác đặc biệt (như char(20)), thì giá trị kết quả sẽ được trình biên dịch của PL/pgSQL ngầm chuyển đổi bằng việc sử dụng hàm đầu ra của dạng kết quả đó và hàm đầu vào của biến đó. Lưu ý rằng điều này có thể tiềm tàng gây ra các lỗi thời gian chạy được hàm đầu vào sinh ra, nếu dạng chuỗi của giá trị kết quả là không được chấp nhận đối với hàm đầu vào. Các ví dụ:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

39.5.2. Thực thi lệnh không có kết quả

Đối với bất kỳ lệnh SQL nào mà không trả về các hàng, ví dụ INSERT mà không có mệnh đề RETURNING, thì bạn có thể thực thi lệnh đó trong một hàm PL/pgSQL chỉ bằng việc viết lệnh đó.

Bất kỳ tên biến PL/pgSQL nào xuất hiện trong văn bản lệnh cũng được đối xử như một tham số, và sau đó giá trị hiện hành của biến đó được cung cấp như là giá trị tham số ở thời điểm chạy. Điều này chính xác giống việc xử lý được mô tả trước đó cho các biểu thức; để có thêm các chi tiết, xem Phần 39.10.1.

Khi thực thi một lệnh SQL theo cách này, PL/pgSQL lên kế hoạch cho lệnh chỉ một lần và sử dụng lại kế hoạch đó trong các thực thi tiếp sau, cho cả đời của kết nối cơ sở dữ liệu. Các tác động của điều này sẽ được thảo luận chi tiết trong Phần 39.10.2.

Đôi khi là hữu dụng để đánh giá một biểu thức hoặc truy vấn SELECT nhưng vứt bỏ kết quả, ví dụ, khi gọi một hàm có các hiệu ứng phụ nhưng không có giá trị kết quả hữu dụng nào. Để làm điều này trong PL/pgSQL, hãy sử dụng lệnh PERFORM:

```
PERFORM query;
```

Điều này thực thi query và vứt bỏ kết quả. Hãy viết query cách y hệt mà bạn có thể viết một lệnh SQL SELECT, nhưng thay thế từ khóa đầu SELECT bằng PERFORM. Đối với các truy vấn WITH, hãy sử dụng PERFORM và sau đó đặt truy vấn đó vào trong các dấu ngoặc đơn. (Trong trường hợp này, truy vấn đó chỉ có thể trả về 1 hàng).

Các biến PL/pgSQL sẽ được thay thế trong truy vấn hệt như cho các lệnh không trả về kết quả, và kế hoạch đó được lưu giữ lại theo cách y hệt. Hơn nữa, biến đặc biệt FOUND được thiết lập về đúng (true) nếu truy vấn đó đã sản xuất ra ít nhất 1 hàng, hoặc sai (false) nếu nó đã sản xuất ra không hàng nào cả (xem Phần 39.5.5).

> **Lưu ý**: Người ta có thể kỳ vọng rằng việc viết SELECT trực tiếp có thể hoàn thành kết quả này, nhưng hiện tại cách được chấp nhận duy nhất để làm điều này là PERFORM. Một lệnh SQL có thể trả về các hàng, như SELECT, sẽ bị từ chối như một lỗi trừ phi nó có một mệnh đề INTO như được thảo luận trong phần tiếp sau.

Một ví dụ:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

39.5.3. Thực thi một truy vấn với kết quả một hàng duy nhất

Kết quả của một lệnh SQL cho một hàng duy nhất (có khả năng nhiều cột) có thể được chỉ định cho một biến bản ghi, biến dạng hàng, hoặc danh sách các biến vô hướng. Điều này được thực hiện bằng việc viết lệnh SQL cơ bản và thêm một mệnh đề INTO. Ví dụ,

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

trong đó target có thể là một biến bản ghi, một biến hàng, hoặc một danh sách các biến đơn giản và các trường bản ghi/hàng được phân cách nhau bằng một dấu phẩy. Các biến PL/pgSQL sẽ được thay thế trong phần còn lại của truy vấn, và kế hoạch đó được lưu giữ lại, hệt như được mô tả ở trên cho các lệnh không trả về hàng nào. Điều này làm việc được cho SELECT, INSERT/UPDATE /DELETE với

RETURNING, và các lệnh tiện ích mà trả về các kết quả tập hợp hàng (như EXPLAIN). Ngoại trừ mệnh đề INTO, lệnh SQL đó là y hệt như nó có thể được viết bên ngoài PL/pgSQL.

> **Mẹo**: Lưu ý rằng sự diễn giải này của SELECT với INTO là hoàn toàn khác với lệnh thông thường của PostgreSQL là SELECT INTO, trong đó đích INTO là một bảng mới được tạo ra. Nếu bạn muốn tạo ra một bảng từ một kết quả của SELECT trong một hàm PL/pgSQL, hãy sử dụng cú pháp CREATE TABLE ... AS SELECT.

Nếu một hàng hoặc một danh sách biến được sử dụng như là đích, thì các cột kết quả của truy vấn phải chính xác khớp cấu trúc của đích đó như đối với các dạng số và dữ liệu, nếu không thì một lỗi thời gian chạy sẽ xảy ra. Khi một biến bản ghi là đích, nó tự động thiết lập cấu hình cho bản thân nó về dạng hàng của các cột kết quả truy vấn.

Mệnh đề INTO có thể xuất hiện hầu như bất kỳ đâu trong lệnh SQL. Thông thường nó được viết hoặc ngay trước hoặc ngay sau danh sách của *select_expressions* trong một lệnh SELECT, hoặc ở cuối của lệnh đó đối với các dạng lệnh khác. Được khuyến cáo rằng bạn tuân theo quy ước này trong trường hợp trình phân tích cú pháp PL/pgSQL trở nên khắt khe hơn trong các phiên bản sau.

Nếu STRICT không được chỉ định trong mệnh đề INTO, thì đích sẽ được thiết lập về hàng đầu được truy vấn đó trả về, hoặc về null nếu truy vấn đó không trả về hàng nào. (Lưu ý rằng "hàng đầu tiên" không được định nghĩa tốt trừ phi bạn đã sử dụng ORDER BY). Bất kỳ các hàng kết quả nào sau hàng đầu tiên sẽ bị vứt bỏ. Bạn có thể kiểm tra biến đặc biệt FOUND (xem Phần 39.5.5) để xác định liệu một hàng đã được trả về hay chưa:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
        RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

Nếu lựa chọn STRICT được chỉ định, thì truy vấn đó phải trả về chính xác một hàng hoặc một lỗi thời gian chạy sẽ được nêu, hoặc NO_DATA_FOUND (không hàng nào) hoặc TOO_MANY_ROWS (hơn một hàng). Bạn có thể sử dụng một khối ngoại lệ nếu bạn muốn bắt được lỗi đó, ví dụ:

```
BEGIN
        SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
        EXCEPTION
                WHEN NO_DATA_FOUND THEN
                        RAISE EXCEPTION 'employee % not found', myname;
                WHEN TOO_MANY_ROWS THEN
                        RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Thực thi thành công lệnh với STRICT luôn đặt FOUND về đúng (true).

Đối với INSERT/UPDATE /DELETE với RETURNING, PL/pgSQL nêu một lỗi cho nhiều hơn 1 hàng được trả về, thậm chí khi STRICT không được chỉ định. Điều này là vì không có lựa chọn nào như ORDER BY với nó để xác định hàng bị tác động nào sẽ được trả về.

> **Lưu ý**: Lựa chọn STRICT khớp hành vi của SELECT INTO của PL/SQL của Oracle và các lệnh có liên quan. Để xử trí các trường hợp nơi mà bạn cần xử lý nhiều hàng kết quả từ một truy vấn SQL, hãy xem Phần 39.6.4.

39.5.4. Thực thi lệnh động

Thường xuyên bạn sẽ muốn sinh ra các lệnh động trong các hàm PL/pgSQL của bạn, đó là, các lệnh sẽ liên quan tới các bảng khác nhau hoặc các dạng dữ liệu khác nhau mỗi lần chúng được thực thi. Các cố gắng thông thường của PL/pgSQL để lưu giữ các kế hoạch cho các lệnh (như được thảo luận trong Phần 39.10.2) sẽ không làm việc trong các kịch bản như vậy. Để xử trí dạng vấn đề này, lệnh EXECUTE được đưa ra:

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

trong đó *command-string* là một biểu thức lấy một chuỗi (hoặc dạng text) chứa lệnh sẽ được thực thi. Tùy chọn *target* là một biến bản ghi, một biến hàng, hoặc một danh sách các biến đơn giản và các trường bản ghi/hàng được phân cách nhau bằng một dấu phẩy, trong đó các kết quả của lệnh sẽ được lưu trữ. Các biểu thức tùy chọn USING cung cấp các giá trị sẽ được chèn vào trong lệnh đó.

Không sự thay thế các biến PL/pgSQL nào được thực hiện trong chuỗi lệnh được tính toán. Bất kỳ giá trị biến được yêu cầu nào cũng phải được chèn vào chuỗi lệnh như nó được cấu trúc; hoặc bạn có thể sử dụng các tham số như được mô tả bên dưới.

Hơn nữa, không có kế hoạch cho việc lưu giữ đối với các lệnh được thực thi qua EXECUTE. Thay vào đó, lệnh được chuẩn bị mỗi lần lệnh đó được chạy. Vì thế chuỗi lệnh đó có thể được tạo động trong hàm đó để thực hiện các hành động trong các bảng và các cột khác.

Mệnh đề INTO chỉ định nơi các kết quả của một lệnh SQL trả về các hàng sẽ được chỉ định. Nếu một danh sách biến hoặc hàng được cung cấp, nó phải khớp chính xác cấu trúc các kết quả của truy vấn (khi một biến bản ghi được sử dụng, nó sẽ thiết lập cấu hình cho bản thân nó để khớp với cấu trúc kết quả một cách tự động). Nếu nhiều hàng được trả về, thì chỉ hàng đầu tiên sẽ được chỉ định cho biến INTO. Nếu không hàng nào được trả về, thì NULL được chỉ định cho (các) biến INTO. Nếu không mệnh đề INTO nào được chỉ định, thì các kết quả truy vấn sẽ bị vứt bỏ.

Nếu lựa chọn STRICT được đưa ra, thì 1 lỗi được nêu trừ phi truy vấn đó tạo ra chính xác 1 hàng.

Chuỗi lệnh đó có thể sử dụng các giá trị tham số, chúng được tham chiếu trong lệnh như là $1, ưu tiên cho việc chèn các giá trị dữ liệu vào chuỗi lệnh như là văn bản: nó tránh được tổng chi phí thời gian chạy của việc chuyển đổi các giá trị sang văn bản và ngược lại, và nó là ít có xu hướng hơn nhiều bị tấn công tiêm SQL vì không có nhu cầu cho việc trích dẫn hoặc thoát ra. một ví dụ là:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
        INTO c
        USING checked_user, checked_date;
```

Lưu ý rằng các biểu tượng tham số chỉ có thể được sử dụng cho các giá trị dữ liệu - nếu bạn muốn sử dụng các tên cột hoặc bảng được xác định động, thì bạn phải chèn chúng vào chuỗi lệnh như các văn bản. Ví dụ, nếu truy vấn trước cần phải được thực hiện đối với một bảng được chọn động, bạn có thể làm điều này:

```
EXECUTE 'SELECT count(*) FROM '
        || tabname::regclass
        || ' WHERE inserted_by = $1 AND inserted <= $2'
        INTO c
```

```
    USING checked_user, checked_date;
```

Một hạn chế khác trong các biểu tượng tham số là chúng chỉ làm việc trong các lệnh SELECT, INSERT, UPDATE, và DELETE. Trong các dạng lệnh khác (thường được gọi là các lệnh tiện ích), bạn phải chèn các giá trị bằng văn bản thậm chí nếu chúng chỉ là các giá trị dữ liệu.

Một EXECUTE với một chuỗi lệnh hằng đơn giản và vài tham số USING, như trong ví dụ đầu ở trên, là tương đương về chức năng với hệt việc viết lệnh đó trực tiếp trong PL/pgSQL và cho phép thay thế các biến của PL/pgSQL để xảy ra một cách tự động. Sự khác biệt quan trọng là EXECUTE sẽ lên kế hoạch lại lệnh đó trong từng sự thực thi, sinh ra một kế hoạch mà là đặc biệt cho các giá trị tham số hiện hành; trong khi PL/pgSQL thường tạo ra một kế hoạch chung và lưu giữ nó để sử dụng lại. Trong các tình huống nơi mà kế hoạch tốt nhất phụ thuộc mạnh vào các giá trị tham số, thì EXECUTE có thể là nhanh hơn đáng kể; trong khi kế hoạch đó không là nhạy cảm với các giá trị tham số, thì việc lên lại kế hoạch sẽ là một sự uổng phí công sức.

SELECT INTO hiện không được hỗ trợ trong EXECUTE; thay vào đó, hãy thực thi một lệnh SELECT và chỉ định INTO như một phần của bản thân EXECUTE.

> **Lưu ý**: Lệnh EXECUTE của PL/pgSQL không có liên quan tới lệnh EXECUTE SQL được máy chủ PostgreSQL hỗ trợ. Lệnh EXECUTE của máy chủ không thể được sử dụng trực tiếp trong các hàm PL/pgSQL (và là không cần thiết).

Ví dụ 39-1. Trích các giá trị trong các truy vấn động

Khi làm việc với các lệnh động, bạn sẽ thường phải xử trí việc thoát ra của các trích dẫn. Phương pháp được khuyến cáo cho việc trích dẫn văn bản cố định trong thân hàm của bạn là đặt giữa các dấu đô la ($). (Nếu bạn có mã di sản mà không sử dụng được việc đặt giữa các dấu $, thì hãy tham chiếu tới tổng quan trong Phần 39.11.1, điều có thể tiết kiệm cho bạn một số nỗ lực khi dịch mã được nói tới một sơ đồ hợp lý hơn).

Các giá trị động mà sẽ được chèn vào trong truy vấn được xây dựng đòi hỏi việc xử trí thận trọng vì có thể bản thân chúng chứa các ký tự trích dẫn. Ví dụ (điều này giả thiết rằng bạn đang sử dụng việc đặt giữa các dấu $ cho hàm đó như một tổng thể, nên các dấu trích dẫn không cần phải đúp bản):

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

Ví dụ này thể hiện sử dụng các hàm quote_ident và quote_literal (xem Phần 9.4). Để an toàn, các biểu thức chứa các mã định danh cột hoặc bảng sẽ được truyền qua quote_ident trước khi chèn vào một truy vấn động. Các biểu thức chứa các giá trị sẽ là các chuỗi hằng trong lệnh được xây dựng sẽ được truyền qua quote_literal. Các hàm đó thực hiện các bước phù hợp để trả về văn bản đầu vào nằm trong các dấu ngoặc kép hoặc đơn một cách tương ứng, với bất kỳ các ký tự đặc biệt nào được nhúng được thoát ra một cách đúng phù hợp.

Vì quote_literal được gắn nhãn STRICT, nó sẽ luôn trả về null khi được gọi với một đối số null. Trong ví dụ ở trên, nếu newvalue hoặc keyvalue từng là null, thì toàn bộ chuỗi truy vấn động đó có thể trở thành null, dẫn tới một lỗi từ EXECUTE. Bạn có thể tránh được vấn đề này bằng việc sử dụng hàm quote_nullable, nó làm việc y hệt như quote_literal ngoại trừ là khi được gọi với một đối số null thì nó trả về chuỗi NULL. Ví dụ:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

Nếu bạn đang làm việc với các giá trị có thể là null, thì bạn thường sẽ sử dụng quote_nullable thay cho quote_literal.

Như thường lệ, sự chăm sóc phải được thực hiện để đảm bảo rằng các giá trị null trong một truy vấn không phân phối các kết quả không được mong đợi. Ví dụ mệnh đề WHERE

```
'WHERE key = ' || quote_nullable(keyvalue)
```

sẽ không bao giờ thành công nếu keyvalue là null, vì kết quả của việc sử dụng toán tử bằng = với một toán hạng null luôn là null. Nếu bạn muốn null làm việc giống như một giá trị khóa thông thường, thì bạn có thể cần viết lại thứ ở trên như là

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(Hiện tại, IS NOT DISTINCT FROM được điều khiển có hiệu quả ít hơn nhiều so với = , nên đừng làm điều này trừ phi bạn phải làm. Xem Phần 9.2 để có thêm thông tin về các null và IS DISTINCT).

Lưu ý rằng việc đưa vào giữa các dấu $ chỉ hữu dụng cho việc trích dẫn văn bản cố định. Có thể là một ý tưởng rất tồi để cố viết ví dụ này như:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

vì nó có thể đứt nếu các nội dung của newvalue không may có chữ $$. Sự phản đối y hệt có thể áp dụng cho bất kỳ dấu phân cách đặt giữa các dấu $ nào khác mà bạn có thể chọn. Vì thế, để trích dẫn văn bản an toàn mà không được biết trước, bạn phải sử dụng quote_literal, quote_nullable, hoặc quote_ident một cách phù hợp.

Một ví dụ lớn hơn nhiều về một lệnh động và EXECUTE có thể thấy trong Ví dụ 39-7, nó xây dựng và thực thi một lệnh CREATE FUNCTION để định nghĩa một hàm mới.

39.5.5. Có tình trạng kết quả

Có vài cách để xác định hiệu quả của lệnh. Phương pháp thứ nhất là sử dụng lệnh GET DIAGNOSTICS, nó có dạng:

```
GET DIAGNOSTICS variable = item [ , ... ];
```

Lệnh này cho phép truy xuất các chỉ số tình trạng hệ thống. Từng item là một từ khóa nhận diện giá trị tình trạng sẽ được phân công cho biến được chỉ định (nó sẽ là dạng dữ liệu đúng để nhận). Các khoản tình trạng hiện hành có sẵn là ROW_COUNT, OID của hàng cuối cùng được lệnh SQL gần đây nhất chèn vào. Lưu ý rằng RESULT_OID chỉ hữu dụng sau một lệnh INSERT trong một bảng có chứa các OID. Ví dụ:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Phương pháp thứ 2 để xác định các hiệu ứng của lệnh là kiểm tra biến đặc biệt có tên là FOUND, nó có dạng boolean. FOUND bắt đầu là sai (false) trong từng lời gọi hàm của PL/pgSQL. Nó được thiết lập theo từng trong các dạng lệnh sau:

- Lệnh SELECT INTO thiết lập FOUND đúng (true) nếu một hàng được phân công, sai nếu không hàng nào được trả về.
- Lệnh PERFORM thiết lập FOUND đúng (true) nếu nó tạo ra (và vứt bỏ) một hoặc nhiều hàng, sai (false) nếu không hàng nào được tạo ra.
- Các lệnh UPDATE, INSERT, và DELETE thiết lập FOUND đúng (true) nếu ít nhất một hàng bị ảnh hưởng, sai (false) nếu không hàng nào bị ảnh hưởng.
- Lệnh FETCH thiết lập FOUND đúng (true) nếu nó trả về một hàng, sai (false) nếu không hàng nào được trả về.
- Lệnh MOVE thiết lập FOUND đúng (true) nếu nó thành công định vị lại con trỏ, sai (false) nếu không.
- Lệnh FOR thiết lập FOUND đúng (true) nếu nó lặp đi lặp lại một hoặc nhiều lần, nếu không thì sai (false). Điều này áp dụng cho tất cả 4 phương án của lệnh FOR (các vòng lặp FOR số nguyên, các vòng lặp FOR thiết lập bản ghi, các vòng lặp FOR thiết lập bản ghi động, và các vòng lặp FOR con trỏ). FOUND được thiết lập theo cách này khi vòng lặp FOR tồn tại; bên trong sự thực thi của vòng lặp, FOUND không được lệnh FOR sửa đổi, dù nó có thể bị thay đổi bằng sự thực thi các lệnh khác bên trong thân vòng lặp đó.
- Các lệnh RETURN QUERY và RETURN QUERY EXECUTE thiết lập FOUND đúng (true) nếu truy vấn trả về ít nhất 1 hàng, sai (false) nếu không hàng nào được trả về.

Các lệnh PL/pgSQL khác không thay đổi tình trạng của FOUND. Lưu ý đặc biệt rằng EXECUTE thay đổi đầu ra của GET DIAGNOSTICS, nhưng không thay đổi FOUND.

FOUND là một biến cục bộ trong từng hàm PL/pgSQL; bất kỳ thay đổi nào đối với nó cũng ảnh hưởng chỉ tới hàm hiện hành.

39.5.6. Hoàn toàn không làm gì cả

Đôi khi một lệnh giữ chỗ không làm gì cả lại là hữu dụng. Ví dụ, nó có thể chỉ ra rằng một nhánh của một chuỗi if/then/else là cố tình để rỗng. Vì lý do này, hãy sử dụng lệnh NULL;:

```
NULL;
```

Ví dụ, 2 đoạn mã sau là tương đương:

```
BEGIN
        y := x / 0;
EXCEPTION
        WHEN division_by_zero THEN
                NULL; -- ignore the error
END;

BEGIN
        y := x / 0;
EXCEPTION
        WHEN division_by_zero THEN -- ignore the error
END;
```

Cái nào được ưa thích hơn là tùy bạn.

> **Lưu ý**: Trong PL/SQL của Oracle, các danh sách lệnh rỗng là không được phép, và vì thế các lệnh NULL được yêu cầu cho các tình huống như thế này. PL/pgSQL cho phép bạn đúng là không viết gì.

39.6. Cấu trúc kiểm tra

Các cấu trúc kiểm tra có thể là phần hữu dụng nhất (và quan trọng) của PL/pgSQL. Với các cấu trúc kiểm tra của PL/pgSQL, bạn có thể xử trí các dữ liệu PostgreSQL theo một cách thức rất mềm dẻo và mạnh mẽ.

39.6.1. Trả về từ một hàm

Có 2 lệnh sẵn sàng cho phép bạn trả về các dữ liệu từ một hàm: RETURN và RETURN NEXT.

39.6.1.1. RETURN

```
RETURN expression;
```

RETURN với một biểu thức kết thúc hàm và trả về giá trị của *expression* cho trình gọi. Dạng này được sử dụng cho các hàm PL/pgSQL mà không trả về một tập hợp.

Khi trả về một dạng vô hướng, bất kỳ biểu thức nào cũng có thể được sử dụng. Kết quả của biểu thức đó sẽ tự động đưa vào trong dạng trả về của hàm đó như được mô tả cho các phân công. Để trả về một giá trị tính tổng (hàng), bạn phải viết một biến hàng hoặc bản ghi như là *expression*.

Nếu bạn đã khai báo hàm với các tham số đầu ra, hãy viết chỉ RETURN không với biểu thức nào. Các giá trị hiện hành của các biến tham số đầu ra sẽ được trả về.

Nếu bạn đã khai báo hàm để trả về void, thì một lệnh RETURN có thể được sử dụng để thoát ra sớm hàm đó; nhưng không viết một biểu thức sau RETURN.

Giá trị trả về của một hàm không thể được để không xác định. Nếu sự kiểm soát tìm kiếm kết thúc của khối mức đỉnh của hàm mà không gặp một lệnh RETURN, thì lỗi thời gian chạy sẽ xảy ra. Tuy nhiên, hạn chế này không áp dụng cho các hàm với các tham số đầu ra và các hàm trả về void. Trong

các trường hợp đó, một lệnh RETURN được thực thi tự động nếu khối mức định kết thúc.

39.6.1.2. RETURN NEXT và RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

Khi một hàm PL/pgSQL được khai báo để trả về SETOF *sometype*, thủ tục để tuân theo là khá khác nhau. Trong trường hợp đó, các khoản riêng rẽ để trả về được tuần tự các lệnh RETURN NEXT hoặc RETURN QUERY phân công, và sau đó một lệnh RETURN cuối cùng không có đối số sẽ được sử dụng để chỉ ra rằng hàm đó đã kết thúc việc thực thi. RETURN NEXT có thể được sử dụng với cả các dạng dữ liệu vô hướng và tổng hợp; với một dạng kết quả tổng hợp, toàn bộ "bảng" các kết quả sẽ được trả về. RETURN QUERY nối thêm các kết quả của việc thực thi một truy vấn tới tập hợp kết quả của hàm đó. RETURN NEXT và RETURN QUERY có thể tự do trộn với nhau trong một hàm trả về tập hợp duy nhất, trong trường hợp đó các kết quả của chúng sẽ được ghép nối nhau.

RETURN NEXT và RETURN QUERY không thực sự trả về từ hàm đó - chúng đơn giản nối thêm zero hoặc nhiều hàng hơn vào tập hợp kết quả của hàm đó. Sự thực thi sau đó tiếp tục với lệnh tiếp sau trong hàm PL/pgSQL. Khi các lệnh tiếp sau RETURN NEXT hoặc RETURN QUERY được thực thi, thì tập hợp kết quả được xây dựng. Một RETURN cuối cùng, sẽ không có đối số, gây ra sự kiểm soát để thoát khỏi hàm đó (hoặc bạn có thể chỉ để sự kiểm soát tới được cuối của hàm đó).

RETURN QUERY có một phương án RETURN QUERY EXECUTE, nó phân công truy vấn đó sẽ thực thi động. Các biểu thức tham số có thể được chèn vào chuỗi truy vấn tính toán qua USING, theo đúng cách thức y hệt như trong lệnh EXECUTE.

Nếu bạn đã khai báo hàm với các tham số đầu ra, hãy chỉ viết RETURN NEXT không có biểu thức gì. Trong từng sự thực thi, các giá trị hiện hành của (các) biến tham số đầu ra sẽ được giữ để trả về như một hàng kết quả. Lưu ý rằng bạn phải khai báo hàm như việc trả về SETOF *record* khi có nhiều tham số đầu ra, hoặc SETOF *sometype* khi có chỉ một tham số đầu ra của dạng *sometype*, để tạo một hàm tự trả về với các tham số đầu ra.

Đây là một ví dụ của một hàm có sử dụng RETURN NEXT:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
        r foo%rowtype;
BEGIN
        FOR r IN SELECT * FROM foo
        WHERE fooid > 0
        LOOP
                -- can do some processing here
                RETURN NEXT r; -- return current row of SELECT
        END LOOP;
        RETURN;
END
$BODY$
```

```
LANGUAGE 'plpgsql' ;

SELECT * FROM getallfoo();
```

> **Lưu ý**: Triển khai hiện hành RETURN NEXT và RETURN QUERY lưu toàn bộ tập hợp kết quả trước khi trả về từ hàm đó, như được thảo luận ở trên. Điều đó có nghĩa là nếu một hàm PL/pgSQL tạo ra một tập hợp kết quả rất rộng, thì hiệu năng có thể là tồi: dữ liệu sẽ được ghi vào đĩa để tránh hết bộ nhớ, nhưng bản thân hàm đó không trả về cho tới khi toàn bộ tập hợp kết quả đã được tạo ra. Một phiên bản trong tương lai của PL/pgSQL có thể cho phép người sử dụng định nghĩa các hàm trả về tập hợp mà không có hạn chế này. Hiện hành, điểm mà ở đó dữ liệu bắt đầu được ghi vào đĩa được biến cấu hình work_mem kiểm soát. Các quản trị viên có bộ nhớ đủ để lưu trữ các tập hợp kết quả lớn hơn trong bộ nhớ nên cân nhắc tăng tham số này.

39.6.2. Thể điều kiện

Các lệnh IF và CASE cho phép bạn thực thi các lệnh lựa chọn dựa vào các điều kiện nhất định. PL/pgSQL có 3 dạng IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE

và 2 dạng của CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

39.6.2.1. IF-THEN

```
IF boolean-expression THEN
        statements
END IF;
```

Các lệnh IF-THEN là dạng đơn giản nhất của IF. Các lệnh giữa THEN và END IF sẽ được thực thi nếu điều kiện là đúng (true). Nếu không, chúng sẽ bị bỏ qua. Ví dụ:

```
IF v_user_id <> 0 THEN
        UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

39.6.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
        statements
ELSE
        statements
END IF;
```

Các lệnh IF-THEN-ELSE thêm vào IF-THEN bằng việc để cho bạn phân công một tập hợp tùy chọn các lệnh sẽ được thực thi nếu điều kiện là đúng (true). (Lưu ý điều này bao gồm trường hợp nơi mà điều kiện đó bằng NULL). Các ví dụ:

```
IF parentid IS NULL OR parentid = ''
THEN
        RETURN fullname;
```

```
ELSE
        RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
        INSERT INTO users_count (count) VALUES (v_count);
        RETURN 't';
ELSE
        RETURN 'f';
END IF;
```

39.6.2.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
        statements
[ ELSIF boolean-expression THEN
        statements
[ ELSIF boolean-expression THEN
        statements
        ...]]
[ ELSE
        statements ]
END IF;
```

Đôi khi có nhiều hơn chỉ 2 lựa chọn. IF-THEN-ELSIF đưa ra một phương pháp kiểm tra thuận tiện vài lựa chọn. Các điều kiện IF được kiểm thử thành công cho tới điều kiện đầu là đúng được tìm thấy. Sau đó (các) lệnh có liên quan sẽ được thực thi, sau đó sự kiểm soát truyền tới lệnh tiếp sau END IF. (Bất kỳ điều kiện IF nào sau đó cũng sẽ không được kiểm thử). Nếu không có điều kiện IF nào là đúng (true), thì khối ELSE (nếu có) sẽ được thực thi.

Đây là một ví dụ:

```
IF number = 0 THEN
        result := 'zero';
ELSIF number > 0 THEN
        result := 'positive';
ELSIF number < 0 THEN
        result := 'negative';
ELSE
        -- hmm, the only other possibility is that number is null
        result := 'NULL';
END IF;
```

Từ khóa ELSIF cũng có thể được đọc là ELSEIF.

Một cách thức chọn lựa của việc hoành thành tác vụ y hệt là lồng các lệnh IF-THEN-ELSE, như trong ví dụ sau:

```
IF demo_row.sex = 'm' THEN
        pretty_sex := 'man';
ELSE
        IF demo_row.sex = 'f' THEN
                pretty_sex := 'woman';
        END IF;
END IF;
```

Tuy nhiên, phương pháp này đòi hỏi việc viết một END IF khớp cho từng IF, nên nó cồng kềnh hơn nhiều so với việc sử dụng ELSIF khi có nhiều lựa chọn thay thế.

39.6.2.4. CASE đơn giản

```
CASE search-expression
        WHEN expression [, expression [ ... ]] THEN
                statements
        [ WHEN expression [, expression [ ... ]] THEN
                statements
                ... ]
        [ ELSE
        statements ]
END CASE;
```

Dạng đơn giản của CASE đưa ra sự thực thi có điều kiện dựa vào sự ngang bằng nhau của các toán tử. *search-expression* được đánh giá (một khi) và được so sánh thành công với từng *expression* trong các mệnh đề WHEN. Nếu một sự trùng khớp được thấy, thì *statements* tương ứng sẽ được thực thi, và sau đó sự kiểm soát truyền sang lệnh tiếp sau END CASE. (Tuần tự các biểu thức WHEN sẽ không được đánh giá). Nếu không sự trùng khớp nào được thấy, thì ELSE statements sẽ được thực thi; nhưng nếu ELSE không hiện diện, thì một ngoại lệ CASE_NOT_FOUND sẽ nảy sinh.

Đây là ví dụ đơn giản:

```
CASE x
        WHEN 1, 2 THEN
                msg := 'one or two';
        ELSE
                msg := 'other value than one or two';
END CASE;
```

39.6.2.5. CASE được tìm kiếm

```
CASE
        WHEN boolean-expression THEN
                statements
        [ WHEN boolean-expression THEN
                statements
                ... ]
        [ ELSE
                statements ]
END CASE;
```

Dạng được tìm kiếm của CASE đưa ra sự thực thi có điều kiện dựa vào sự đúng đắn của các biểu thức Boolean. Từng *boolean-expression* của mệnh đề WHEN được đánh giá lần lượt, cho tới khi một thứ được thấy là true. Sau đó statements tương ứng sẽ được thực thi, và sau đó sự kiểm soát truyền sang lệnh tiếp sau END CASE. (Các biểu thức WHEN tiếp sau sẽ không được đánh giá). Nếu không kết quả đúng nào được thấy, thì ELSE statements sẽ được thực thi; nhưng nếu ELSE không hiện diện, thì một biểu thức CASE_NOT_FOUND sẽ nảy sinh.

Đây là một ví dụ:

```
CASE
        WHEN x BETWEEN 0 AND 10 THEN
                msg := 'value is between zero and ten';
        WHEN x BETWEEN 11 AND 20 THEN
                msg := 'value is between eleven and twenty';
END CASE;
```

Dạng CASE này hoàn toàn tương đương với IF-THEN-ELSIF, ngoại trừ quy tắc đạt tới một mệnh đề

ELSE bị bỏ qua gây ra một lỗi thay vì không làm gì.

39.6.3. Vòng lặp đơn giản

Với các lệnh LOOP, EXIT, CONTINUE, WHILE, và FOR, bạn có thể sắp xếp cho hàm PL/pgSQL của bạn để lặp lại một loạt các lệnh.

39.6.3.1. LOOP

```
[ <<label>> ]
LOOP
        statements
END LOOP [ label ];
```

LOOP định nghĩa một vòng lặp vô điều kiện được lặp lại bất tận cho tới khi một lệnh EXIT hoặc RETURN kết thúc nó. Nhãn tùy chọn label có thể được các lệnh EXIT và CONTINUE sử dụng trong các vòng lặp lồng nhau để chỉ định vòng lặp nào các lệnh đó tham chiếu tới.

39.6.3.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

Nếu không nhãn label nào được đưa ra, thì vòng lặp bên trong cùng sẽ kết thúc và lệnh đi sau END LOOP được thực thi tiếp sau. Nếu *label* được đưa ra, thì nó phải là nhãn của mức hiện hành hoặc vài mức ngoài của vòng lặp hoặc khối lồng nhau. Sau đó vòng lặp hoặc khối có tên đó được kết thúc và sự kiểm soát tiếp tục với lệnh sau END tương ứng của vòng lặp/khối đó.

Nếu WHEN được chỉ định, thì sự thoát vòng lặp chỉ xảy ra nếu *boolean-expression* là đúng. Nếu không, sự kiểm soát truyền sang lệnh sau EXIT.

EXIT có thể được sử dụng với tất cả các dạng các vòng lặp; không bị hạn chế để sử dụng với các vòng lặp vô điều kiện.

Khi được sử dụng với một khối BEGIN, EXIT truyền sự kiểm soát tới lệnh tiếp sau sự kết thúc của khối đó. Lưu ý là một nhãn phải được sử dụng cho mục đích này; một EXIT không có nhãn không bao giờ được xem xét để khớp với một khối BEGIN. (Đây là một sự thay đổi từ các phiên bản trước 8.4 của PostgreSQL, nó có thể cho phép một EXIT không có nhãn khớp với một khối BEGIN).

Các ví dụ:

```
LOOP
        -- some computations
        IF count > 0 THEN
                EXIT; -- exit loop
        END IF;
END LOOP;

LOOP
        -- some computations
        EXIT WHEN count > 0; -- same result as previous example
END LOOP;

<<ablock>>
BEGIN
        -- some computations
        IF stocks > 100000 THEN
```

```
            EXIT ablock; -- causes exit from the BEGIN block
        END IF;
        -- computations here will be skipped when stocks > 100000
END;
```

39.6.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

Nếu không nhãn *label* nào được đưa ra, sự lặp đi lặp lại tiếp sau của vòng lặp bên trong cùng được bắt đầu. Đó là, tất cả các lệnh còn lại trong thân vòng lặp đó sẽ bị bỏ qua, và sự kiểm soát trả về cho vòng lặp đó biểu thức kiểm soát (nếu có) để xác định liệu sự lặp đi lặp lại của vòng lặp khác có là cần thiết hay không. Nếu *label* hiện diện, nó chỉ định nhãn của vòng lặp mà sự thực thi của nó sẽ được tiếp tục.

Nếu WHEN được chỉ định, thì sự lặp đi lặp lại tiếp sau của vòng lặp chỉ được bắt đầu nếu *boolean-expression* là đúng. Nếu không, sự kiểm soát truyền tới lệnh sau CONTINUE.

CONTINUE có thể được sử dụng với tất cả các dạng vòng lặp; không có hạn chế nào để sử dụng với các vòng lặp vô điều kiện.

Các ví dụ:

```
LOOP
        -- some computations
        EXIT WHEN count > 100;
        CONTINUE WHEN count < 50;
        -- some computations for count IN [50 .. 100]
END LOOP;
```

39.6.3.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
        statements
END LOOP [ label ];
```

Lệnh WHILE lặp một tuần tự các lệnh miễn là *boolean-expression* ước tính về đúng (true). Biểu thức được kiểm tra ngay trước khi từng khoản đầu vào tới thân vòng lặp. Ví dụ:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
        -- some computations here
END LOOP;

WHILE NOT done LOOP
        -- some computations here
END LOOP;
```

39.6.3.5. FOR (phương án số nguyên)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
        statements
END LOOP [ label ];
```

Dạng này của FOR tạo vòng lặp mà lặp đi lặp lại đối với một dãy các giá trị số nguyên. Biến name tự

động được xác định như dạng integer và chỉ tồn tại bên trong vòng lặp (bất kỳ định nghĩa đang tồn tại nào của tên biến đó cũng bị bỏ qua với vòng lặp đó). 2 biểu thức trao ràng buộc thấp hơn và cao hơn của dãy đó sẽ được đánh giá một lần khi vào vòng lặp. Nếu mệnh đề BY không được chỉ định thì bước lặp là 1, nếu không thì đó là giá trị được chỉ định trong mệnh đề BY, nó một lần nữa được đánh giá một lần trong khoản đầu vào của vòng lặp. Nếu REVERSE được chỉ định thì giá trị bước đó được trừ đi, thay vì được cộng thêm vào, sau từng sự lặp.

Vài ví dụ của các vòng lặp FOR số nguyên:

```
FOR i IN 1..10 LOOP
        -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
        -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
        -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

Nếu ràng buộc thấp hơn là lớn hơn so với ràng buộc cao hơn (hoặc ít hơn, trong trường hợp REVERSE), thì thân vòng lặp không được thực thi hoàn toàn. Không có lỗi nào xảy ra.

Nếu một label được gắn vào vòng lặp FOR thì biến vòng lặp số nguyên đó có thể được tham chiếu với một tên có đủ điều kiện, bằng việc sử dụng *label*.

39.6.4. Lặp qua kết quả truy vấn

Sử dụng dạng khác nhau của vòng lặp FOR, bạn có thể lặp qua các kết quả của một truy vấn và xử trí dữ liệu đó một cách tương xứng. Cú pháp là:

```
[ <<label>> ]
FOR target IN query LOOP
        statements
END LOOP [ label ];
```

target là một biến bản ghi, biến hàng, hoặc danh sách các biến vô hướng được phân cách nhau bằng dấu phẩy. *target* được chỉ định thành công cho từng hàng kết quả từ truy vấn *query* và thân vòng lặp được thực thi cho từng hàng. Đây là một ví dụ:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
        mviews RECORD;
BEGIN
        PERFORM cs_log('Refreshing materialized views...');
        FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
                -- Now "mviews" has one record from cs_materialized_views

                PERFORM cs_log('Refreshing materialized view '
                        || quote_ident(mviews.mv_name) || ' ...');
                EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
                EXECUTE 'INSERT INTO '
                        || quote_ident(mviews.mv_name) || ' '
                        || mviews.mv_query;
        END LOOP;
        PERFORM cs_log('Done refreshing materialized views.');
```

```
        RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Nếu vòng lặp được một lệnh EXIT kết thúc, thì giá trị hàng được chỉ định cuối cùng vẫn truy cập được sau vòng lặp đó.

query được sử dụng trong dạng lệnh FOR này có thể là bất kỳ lệnh SQL nào mà trả về các hàng cho trình gọi: SELECT là trường hợp phổ biến nhất, nhưng bạn cũng có thể sử dụng INSERT, UPDATE, hoặc DELETE với một mệnh đề RETURNING. Vài lệnh tiện ích như EXPLAIN cũng sẽ làm việc.

Các biến PL/pgSQL được thay thế trong văn bản truy vấn, và kế hoạch truy vấn được lưu giữ để có khả năng sử dụng lại, như được thảo luận chi tiết trong Phần 39.10.1 và Phần 39.10.2.

Lệnh FOR-IN-EXECUTE là cách khác để lặp đối với các hàng:

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
        statements
END LOOP [ label ];
```

Điều này giống như dạng trước, ngoại trừ là truy vấn nguồn được chỉ định như một biểu thức chuỗi, nó được đánh giá và lên kế hoạch lại trong từng khoản đầu vào đối với vòng lặp FOR. Điều này cho phép lập trình viên chọn tốc độ của một truy vấn được lên kế hoạch sẵn hoặc tính mềm dẻo của một truy vấn động, hệt như với một lệnh EXECUTE thông thường. Với EXECUTE, các giá trị tham số có thể được chèn vào lệnh động qua USING.

Một cách khác để chỉ định truy vấn mà các kết quả của nó sẽ được lặp là khai báo nó như một con trỏ. Điều này được mô tả trong Phần 39.7.4.

39.6.5. Bắt lỗi

Mặc định, bất kỳ lỗi nào xảy ra trong một hàm PL/pgSQL cũng bỏ sự thực thi của hàm, và quả thực cũng của giao dịch xung quanh nữa. Bạn có thể bắt các lỗi và phục hồi từ chúng bằng việc sử dụng một khối BEGIN với một mệnh đề EXCEPTION. Cú pháp là một mở rộng của cú pháp thông thường cho một khối BEGIN:

```
[ <<label>> ]
[ DECLARE
        declarations ]
BEGIN
        statements
EXCEPTION
        WHEN condition [ OR condition ... ] THEN
                handler_statements
        [ WHEN condition [ OR condition ... ] THEN
                handler_statements
        ... ]
END;
```

Nếu không lỗi nào xảy ra, thì dạng khối này đơn giản thực thi tất cả các *statements*, và sau đó sự kiểm soát truyền tới lệnh tiếp sau END. Nhưng nếu một lỗi xảy ra trong *statements*, thì việc xử lý tiếp *statements* sẽ bị bỏ, và sự kiểm soát truyền tới danh sách EXCEPTION. Danh sách đó được tìm kiếm điều kiện *condition* đầu tiên khớp với lỗi đã xảy ra. Nếu một sự trùng khớp được thấy, thì

handler_statements tương ứng sẽ được thực thi, và sau đó sự kiểm soát truyền tới lệnh tiếp sau END. Nếu không sự khớp nào được thấy, thì lỗi sẽ phát sinh dù mệnh đề EXCEPTION không còn ở đó nữa hoàn toàn: lỗi đó có thể bị bắt bằng một khối vây quanh EXCEPTION, hoặc nếu không có thì nó bỏ qui trình của hàm đó.

Các tên *condition* có thể là bất kỳ như được chỉ ra trong Phụ lục A. Tên một chủng loại khớp bất kỳ lỗi nào trong chủng loại của nó. Tên điều kiện đặc biệt OTHERS khớp mọi dạng error ngoại trừ QUERY_CANCELED. (Có khả năng, nhưng thường không khôn ngoan, để bẫy QUERY_CANCELED bằng tên). Các tên điều kiện không phân biệt chữ hoa chữ thường. Hơn nữa, một điều kiện lỗi có thể được mã SQLSTATE phân công; ví dụ những điều sau là tương đương:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Nếu một lỗi mới xảy ra trong *handler_statements* được lựa chọn, thì nó có thể bị mệnh đề EXCEPTION này bắt được, nhưng bị lan truyền ra. Một mệnh đề EXCEPTION bao quanh có thể bắt nó. Khi một lỗi bị một mệnh đề EXCEPTION bắt được, các biến cục bộ của hàm PL/pgSQL vẫn giữ nguyên như chúng đã từng khi lỗi đã xảy ra, nhưng tất cả các thay đổi tới tình trạng cơ sở dữ liệu nhất quán trong khối đó sẽ được quay ngược lại như cũ. Như một ví dụ, hãy xem xét đoạn này:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
        UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
        x := x + 1;
        y := x / 0;
EXCEPTION
        WHEN division_by_zero THEN
                RAISE NOTICE 'caught division_by_zero';
                RETURN x;
END;
```

Khi kiểm soát đạt tới được y, thì nó sẽ hỏng với lỗi division_by_zero. Điều này sẽ được mệnh đề EXCEPTION bắt được. Giá trị được trả về trong lệnh RETURN sẽ là giá trị tăng dần của x, nhưng các hiệu ứng của lệnh UPDATE sẽ được quay ngược về như cũ. Tuy nhiên, lệnh INSERT đứng trước khối đó sẽ không quay ngược, nên kết quả cuối cùng là cơ sở dữ liệu đó chứa Tom Jones chứ không phải Joe Jones.

> **Mẹo**: Một khối chứa mệnh đề EXCEPTION là đắt hơn đáng kể để vào và ra hơn khối không chứa nó. Vì thế, đừng sử dụng EXCEPTION nếu không cần.

Trong một trình điều khiển ngoại lệ, biến SQLSTATE chứa mã lỗi tương ứng với ngoại lệ mà đã phát sinh (tham chiếu tới Bảng A-1 đối với một danh sách các mã lỗi có khả năng). Biến SQLERRM chứa thông điệp lỗi có liên quan tới ngoại lệ đó. Các biến đó không được xác định bên ngoài các trình điều khiển ngoại lệ.

Ví dụ 39-2. Các ngoại lệ với UPDATE/INSERT

Ví dụ này sử dụng điều khiển ngoại lệ để thực hiện hoặc UPDATE hoặc INSERT một cách phù hợp:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);
```

```
CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
        LOOP
                -- first try to update the key
                UPDATE db SET b = data WHERE a = key;
                IF found THEN
                        RETURN;
                END IF;
                -- not there, so try to insert the key
                -- if someone else inserts the same key concurrently,
                -- we could get a unique-key failure
                BEGIN
                        INSERT INTO db(a,b) VALUES (key, data);
                        RETURN;
                EXCEPTION WHEN unique_violation THEN
                        -- do nothing, and loop to try the UPDATE again
                END;
        END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

39.7. Con trỏ

Thay vì thực thi toàn bộ một truy vấn một lần, là có khả năng để thiết lập một *con trỏ* (*cursor*) gói gọn truy vấn đó, và sau đó đọc kết quả truy vấn đó vài hàng một lần. Một lý do để làm điều này là để tránh chạy hết bộ nhớ khi kết quả chứa một số lượng lớn các hàng. (Tuy nhiên, người sử dụng PL/pgSQL thường không cần phải lo lắng về điều đó, vì các vòng lặp FOR tự động sử dụng một con trỏ để tránh các vấn đề bộ nhớ trong nội bộ). Một ứng dụng thú vị hơn là trả về một tham chiếu tới một con trỏ mà một hàm đã tạo ra, cho phép trình gọi đọc các hàng. Điều này đưa ra một cách hiệu quả để trả về các tập hợp hàng lớn từ các hàm.

39.7.1. Khai báo các biến con trỏ

Tất cả sự truy cập tới các con trỏ trong PL/pgSQL đi qua các biến con trỏ, chúng luôn có dạng dữ liệu đặc biệt refcursor. Một cách để tạo một biến con trỏ là chỉ khai báo nó như một biến có dạng refcursor. Một cách khác là sử dụng cú pháp khai báo con trỏ, thường là:

name [[NO] SCROLL] CURSOR [(*arguments*)] FOR *query*;

(FOR có thể được thay thế bằng IS vì tính tương thích của Oracle). Nếu SCROLL được chỉ định, con trỏ sẽ có khả năng cuộn ngược lại; nếu NO SCROLL được chỉ định, sự nạp ngược sẽ bị từ chối; nếu không đặc tả nào xuất hiện, thì đó là phụ thuộc truy vấn dù nạp ngược sẽ được phép. *arguments*, nếu được chỉ định, là một danh sách các cặp *name datatype* được phân cách nhau bằng dấu phẩy mà định nghĩa các tên sẽ bị các giá trị tham số thay thế trong truy vấn được đưa ra. Các giá trị thực để thay thế cho các tên đó sẽ được chỉ định sau này, khi con trỏ được mở.

Vài ví dụ:

```
DECLARE
        curs1 refcursor;
        curs2 CURSOR FOR SELECT * FROM tenk1;
        curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Cả 3 biến đó có dạng dữ liệu refcursor, nhưng biến đầu có thể được sử dụng với bất kỳ truy vấn nào, trong khi biến thứ 2 có một truy vấn được chỉ định đầy đủ có ràng buộc rồi với nó, và biến cuối cùng có truy vấn được tham số hóa có *ràng buộc (bound)* tới nó. (key sẽ được một giá trị tham số là số nguyên thay thế khi con trỏ được mở). Biến curs1 được nói *không bị ràng buộc (unbound)* vì nó không bị ràng buộc tới bất kỳ truy vấn đặc biệt nào.

39.7.2. Mở con trỏ

Trước khi một con trỏ có thể được sử dụng để truy xuất các hàng, nó phải được *mở*. (Đây là hành động tương đương với lệnh SQL DECLARE CURSOR). PL/pgSQL có 3 dạng lệnh OPEN, 2 trong số đó sử dụng các biến con trỏ không ràng buộc trong khi cái thứ 3 sử dụng một biến con trỏ ràng buộc.

> **Lưu ý**: Các biến con trỏ ràng buộc cũng có thể được sử dụng mà không mở rõ ràng con trỏ, qua lệnh FOR được mô tả trong Phần 39.7.4.

39.7.2.1. Truy vấn OPEN FOR

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

Biến con trỏ được mở và đưa ra truy vấn được phân công để thực thi. Con trỏ đó không thể được mở rồi, và nó phải đã được khai báo như một biến con trỏ không ràng buộc (đó là, như một biến đơn giản refcursor). Truy vấn đó phải là một SELECT, hoặc thứ gì đó mà trả về các hàng (như EXPLAIN). Truy vấn đó được đối xử theo cách thức y hệt như các lệnh SQL khác trong PL/pgSQL: các tên biến của PL/pgSQL được thay thế, và kế hoạch truy vấn được giữ để có thể sử dụng lại.

Khi một biến PL/pgSQL được thay thế trong truy vấn của con trỏ, thì giá trị được thay thế là giá trị nó có vào thời điểm OPEN; những thay đổi sau đó đối với biến đó sẽ không ảnh hưởng tới hành vi của con trỏ. Các lựa chọn SCROLL và NO SCROLL có nghĩa y hệt như đối với một con trỏ ràng buộc.

Ví dụ:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

39.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
[ USING expression [, ... ] ];
```

Biến con trỏ được mở và đưa ra truy vấn được phân công để thực thi. Con trỏ đó không thể là mở rồi, và nó phải đã được khai báo như một biến con trỏ không ràng buộc (đó là, như một biến đơn giản refcursor). Truy vấn đó được chỉ định như một biểu thức chuỗi, theo cách thức y hệt như trong lệnh EXECUTE. Như thường lệ, điều này trao tính mềm dẻo mà kế hoạch truy vấn có thể biến đổi từ kế hoạch này sang kế hoạch khác (xem Phần 39.10.2), và nó cũng có nghĩa là sự thay đổi biến không được thực hiện trong chuỗi lệnh. Như với EXECUTE, các giá trị tham số có thể được chèn vào trong lệnh động qua USING. Các lựa chọn SCROLL và NO SCROLL có cùng ý nghĩa như đối với một

con trỏ ràng buộc. Ví dụ:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
                                        || ' WHERE col1 = $1' USING keyvalue;
```

Trong ví dụ này, tên bảng được chèn vào trong truy vấn dạng văn bản, nên hãy sử dụng quote_ident() được khuyến cáo để canh phòng chống tiêm SQL. Giá trị so sánh cho col1 được chèn vào qua tham số USING, nên nó không cần đưa vào các dấu trích dẫn.

39.7.2.3. Mở con trỏ ràng buộc

OPEN *bound_cursorvar* [(*argument_values*)];

Dạng OPEN này được sử dụng để mở một biến con trỏ mà truy vấn của nó đã bị ràng buộc tới nó khi nó đã được khai báo. Con trỏ đó không thể mở rồi. Một danh sách các biểu thức giá trị đối số thực phải xuất hiện nếu và chỉ nếu con trỏ đó đã được khai báo để lấy các đối số. Các giá trị đó sẽ bị thay thế trong truy vấn đó. Kế hoạch truy vấn cho một con trỏ ràng buộc luôn được xem xét có khả năng giữ lại; không có tương đương của EXECUTE trong trường hợp này. Lưu ý rằng SCROLL và NO SCROLL không thể được chỉ định, khi mà hành vi cuộn của con trỏ đã được xác định rồi.

Lưu ý rằng vì sự thay thế biến được thực hiện trong truy vấn của con trỏ ràng buộc, có 2 cách để truyền các giá trị vào con trỏ đó: hoặc với một đối số rõ ràng cho OPEN, hoặc ngầm bằng việc tham chiếu một biến của PL/pgSQL trong truy vấn đó. Tuy nhiên, chỉ các biến được khai báo trước con trỏ ràng buộc đã được khai báo sẽ được thay thế trong nó. Trong cả 2 trường hợp thì giá trị được truyền được xác định ở thời điểm của OPEN. Các ví dụ:

```
OPEN curs2;
OPEN curs3(42);
```

39.7.3. Sử dụng con trỏ

Một khi một con trỏ đã được mở, nó có thể bị điều khiển bằng các lệnh được mô tả ở đây.

Các thao tác đó cần không xảy ra trong hàm y hệt mà đã mở con trỏ đó để bắt đầu. Bạn có thể trả về một giá trị refcursor nằm ngoài một hàm và để trình gọi vận hành trong con trỏ đó. (Về nội bộ, một giá trị refcursor đơn giản là tên chuỗi của một cổng có chứa truy vấn tích cực cho con trỏ đó. Tên này có thể được truyền quanh, được chỉ định tới các biến refcursor khác, và cứ như thế, không làm phiền cổng đó),

Tất cả các cổng ngầm được đóng lại ở cuối giao dịch. Vì thế một giá trị refcursor là sử dụng được để tham chiếu một con trỏ mở chỉ tới khi kết thúc giao dịch đó.

39.7.3.1. FETCH

FETCH [*direction* { FROM | IN }] *cursor* INTO *target*;

FETCH truy xuất hàng tiếp sau từ con trỏ trong một đích, nó có thể là một biến hàng, một biến bản ghi, hoặc một danh sách các biến đơn giản được phân cách nhau bằng dấu phẩy, hệt như SELECT INTO. Nếu không có hàng tiếp sau, thì đích được thiết đặt về (các) NULL. Như với SELECT INTO, biến đặc biệt FOUND có thể được kiểm tra để thấy liệu đã có được một hàng hay chưa.

Mệnh đề *direction* có thể là bất kỳ phương án nào được phép trong lệnh SQL FETCH ngoại trừ các phương án có thể lấy về hơn 1 hàng; ấy là, nó có thể là NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*,

RELATIVE *count*, FORWARD, hoặc BACKWARD. Việc bỏ qua *direction* là y hệt như việc chỉ định NEXT. Các giá trị *direction* mà đòi hỏi việc dịch chuyển ngược có khả năng thất bại trừ phi con trỏ đã được khai báo hoặc được mở với lựa chọn SCROLL.

cursor phải là tên của một biến refcursor mà tham chiếu tới một cổng con trỏ mở. Các ví dụ:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

39.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE đặt nguyên chỗ cho một con trỏ mà không có việc truy xuất bất kỳ dữ liệu nào. MOVE làm việc chính xác như lệnh FETCH, ngoại trừ nó chỉ đặt nguyên chỗ cho con trỏ và không trả về hàng được dịch chuyển tới. Như với SELECT INTO, biến đặc biệt FOUND có thể được kiểm tra để thấy liệu đã có một hàng tiếp sau phải dịch chuyển tới hay chưa.

Mệnh đề *direction* có thể là bất kỳ phương án nào được phép trong lệnh SQL FETCH, ấy là NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, ALL, FORWARD [*count* | ALL], hoặc BACKWARD [*count* | ALL]. Việc bỏ qua *direction* là y hệt như việc chỉ định NEXT. Các giá trị direction mà đòi hỏi việc dịch chuyển ngược có khả năng thất bại trừ phi con trỏ đã được khai báo hoặc được mở với lựa chọn SCROLL. Các ví dụ:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

39.7.3.3. UPDATE/DELÊT WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Khi một con trỏ được đặt nguyên chỗ trong một hàng của bảng, hàng đó có thể được cập nhật hoặc bị xóa bằng việc sử dụng con trỏ đó để nhận diện hàng đó. Có những hạn chế trong những gì truy vấn của con trỏ đó có thể là (đặc biệt, không lập nhóm) và là tốt nhất để sử dụng FOR UPDATE trong con trỏ đó. Để có thêm thông tin, xem trang tham chiếu DECLARE. Ví dụ:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

39.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE đóng cổng (portal) nằm bên dưới một con trỏ mở. Điều này có thể được sử dụng để phát hành các tài nguyên trước đó. Một ví dụ:

```
CLOSE curs1;
```

39.7.3.5. Trả về con trỏ

Các hàm PL/pgSQL có thể trả về các con trỏ cho trình gọi. Điều này là hữu dụng để trả về nhiều hàng hoặc cột, đặc biệt với các tập hợp kết quả rất lớn. Để làm điều này, hàm mở con trỏ và trả về tên con trỏ cho trình gọi (hoặc đơn giản mở con trỏ bằng việc sử dụng một tên cổng [portal] được chỉ định bởi hoặc nếu không thì tới trình gọi). Trình gọi có thể sau đó lấy các hàng từ con trỏ đó. Con trỏ có thể bị trình gọi đóng lại, hoặc nó sẽ bị đóng tự động khi giao dịch đóng lại.

Tên cổng (portal) được sử dụng cho một con trỏ có thể được lập trình viên chỉ định hoặc được tự động sinh ra. Để chỉ định một tên cổng, đơn giản hãy chỉ định một chuỗi tới biến refcursor trước khi mở nó. Giá trị chuỗi biến refcursor sẽ được OPEN sử dụng như là tên của cổng nằm bên dưới. Tuy nhiên, nếu biến refcursor là null, thì OPEN tự động sinh ra tên mà không xung đột với bất kỳ cổng đang tồn tại nào, và chỉ định nó cho biến refcursor.

> **Lưu ý**: Một biến con trỏ ràng buộc được khởi tạo cho giá trị chuỗi đại diện cho tên của nó, sao cho tên cổng là y hệt như tên biến con trỏ, trừ phi lập trình viên viết đè nó bằng sự chỉ định trước khi mở con trỏ đó. Nhưng một biến con trỏ không ràng buộc mặc định về giá trị null lúc đầu, nên nó sẽ nhận một tên duy nhất được tự động sinh ra, trừ phi bị ghi đè.

Ví dụ sau chỉ ra cách thức một tên con trỏ có thể được trình gọi cung cấp:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
        OPEN $1 FOR SELECT col FROM test;
        RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

Ví dụ sau sử dụng sinh tên con trỏ tự động:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
        ref refcursor;
BEGIN
        OPEN ref FOR SELECT col FROM test;
        RETURN ref;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT reffunc2();

        reffunc2
------------------------------
<unnamed cursor 1>
(1 row)
```

```
FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

Ví dụ sau đây chỉ ra một cách để trả về nhiều con trỏ từ một hàm duy nhất:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
        OPEN $1 FOR SELECT * FROM table_1;
        RETURN NEXT $1;
        OPEN $2 FOR SELECT * FROM table_2;
        RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

39.7.4. Lặp qua kết quả một con trỏ

Có một phương án lệnh FOR cho phép việc lặp đi lặp lại qua các hàng được một con trỏ trả về. Cú pháp là:

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( argument_values ) ] LOOP
        statements
END LOOP [ label ];
```

Biến con trỏ phải bị ràng buộc tới vài truy vấn khi nó đã được khai báo, và nó không thể là mở rồi. Lệnh FOR tự động mở con trỏ đó, và nó đóng con trỏ đó một lần nữa khi vòng lặp thoát. Một danh sách các biểu thức giá trị đối số thực phải xuất hiện nếu và chỉ nếu con trỏ đó đã được khai báo để lấy các đối số. Các giá trị đó sẽ được thay thế trong truy vấn đó, y hệt cách như trong quá trình của một OPEN. Biến *recordvar* tự động được định nghĩa như dạng record và thoát ra chỉ bên trong vòng lặp đó (bất kỳ định nghĩa đang tồn tại nào của tên biến đó sẽ bị bỏ qua trong vòng lặp đó). Mỗi hàng được con trỏ trả về được chỉ định lần lượt cho biến bản ghi này và thân vòng lặp được thực thi.

39.8. Lỗi và thông điệp

Hãy sử dụng lệnh RAISE để nêu các thông điệp và đưa ra các lỗi.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

Lựa chọn *level* chỉ định độ nghiêm trọng của lỗi. Các mức được phép là DEBUG, LOG, INFO, NOTICE, WARNING, và EXCEPTION, với EXCEPTION đang là mặc định. EXCEPTION đưa ra một lỗi (thường thì nó

bỏ qua giao dịch hiện hành); các mức khác chỉ sinh ra các thông điệp các mức ưu tiên khác nhau. Liệu các thông điệp của một ưu tiên đặc biệt có được nêu cho máy trạm hay không, được ghi vào lưu ký máy chủ, hoặc cả 2 được các biến cấu hình log_min_messages và client_min_messages kiểm soát. Xem Chương 18 để có thêm thông tin.

Sau *level* nếu có, bạn có thể viết một *format* (nó phải là một hằng chuỗi đơn giản, không là một biểu thức). Định dạng đó chỉ định văn bản thông điệp lỗi sẽ được nêu. Chuỗi định dạng đó có thể được các biểu thức đối số tùy chọn đi theo và sẽ được chèn vào trong thông điệp đó. Bên trong chuỗi định dạng đó, % được thay thế bằng đại diện chuỗi của giá trị đối số tùy chọn tiếp theo. Hãy viết %% để có một hằng %.

Trong ví dụ này, giá trị của v_job_id sẽ thay thế % trong chuỗi:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

Bạn có thể gắn thông tin bổ sung thêm vào báo cáo lỗi bằng việc viết USING theo sau là các khoản *option = expression*. Các từ khóa *option* được phép là MESSAGE, DETAIL, HINT, và ERRCODE, trong khi từng *expression* có thể là bất kỳ biểu thức có giá trị chuỗi nào. MESSAGE thiết lập văn bản thông điệp lỗi (lựa chọn này không thể được sử dụng ở dạng của RAISE mà bao gồm một chuỗi định dạng trước USING). DETAIL cung cấp một thông điệp chi tiết lỗi, trong khi HINT cung cấp một thông điệp gợi ý. ERRCODE chỉ định mã lỗi (SQLSTATE) cho báo cáo, hoặc theo tên điều kiện như được nêu trong Phụ lục A, hoặc trực tiếp như một mã SQLSTATE với 5 ký tự.

Ví dụ này sẽ bỏ giao dịch với thông điệp và gợi ý lỗi được đưa ra:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
        USING HINT = 'Please check your user id';
```

Hai ví dụ này chỉ các cách tương đương khi thiết lập SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

Có cú pháp RAISE thứ 2 theo đó đối số chính là tên điều kiện hoặc SQLSTATE sẽ được nêu, ví dụ:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Trong cú pháp này, USING có thể được sử dụng để cung cấp một thông điệp lỗi theo thông lệ, chi tiết, hoặc gợi ý. Một cách khác để làm ví dụ sớm là

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

Vẫn còn phương án khác là viết RAISE USING hoặc RAISE *level* USING và đặt mọi thứ vào danh sách USING.

Phương án cuối của RAISE hoàn toàn không có tham số. Dạng này chỉ có thể được sử dụng bên trong một mệnh đề EXCEPTION của một khối BEGIN; nó gây ra lỗi hiện đang được điều khiển để được ném tới khối bao quanh tiếp sau.

Nếu không tên điều kiện nào, cũng không SQLSTATE được chỉ định trong một lệnh RAISE EXCEPTION, thì mặc định là sử dụng RAISE_EXCEPTION (P0001). Nếu không văn bản thông điệp nào được chỉ

định, thì mặc định là sử dụng tên điều kiện hoặc SQLSTATE như là văn bản thông điệp.

> **Lưu ý**: Khi việc chỉ định một mã lỗi bằng mã SQLSTATE, bạn không bị hạn chế đối với các mã lỗi được xác định sẵn, nhưng có thể chọn bất kỳ mã lỗi nào cấu tạo từ 5 chữ số và/hoặc các ký tự ASCII viết hoa, khác với 00000. Được khuyến cáo rằng bạn tránh đưa ra các mã lỗi mà kết thúc trong 3 số zero, vì có các mã chủng loại như vậy và chỉ có thể bắt được bằng việc bẫy toàn bộ chủng loại đó.

3.9. Thủ tục trigger

PL/pgSQL có thể được sử dụng để định nghĩa các thủ tục trigger. Một thủ tục trigger được tạo ra bằng lệnh CREATE FUNCTION, khai báo nó như một hàm không có các đối số và dạng trả về trigger.

Lưu ý rằng hàm đó phải được khai báo không có đối số thậm chí nếu nó kỳ vọng nhận được các đối số được chỉ định trong CREATE TRIGGER - các đối số trigger được truyền qua TG_ARGV, như được mô tả bên dưới.

Khi một hàm PL/pgSQL được gọi như một trigger, vài biến đặc biệt được tạo ra tự động trong khối mức đỉnh. Chúng là:

NEW

> Dạng dữ liệu RECORD; biến nắm giữ hàng cơ sở dữ liệu mới cho các hoạt động INSERT /UPDATE trong các trigger mức hàng. Biến này là NULL trong các trigger mức lệnh và cho các hoạt động DELETE.

OLD

> Dạng dữ liệu RECORD; biến nắm giữ hàng cơ sở dữ liệu cũ cho các hoạt động UPDATE /DELETE trong các trigger mức hàng. Biến này bằng NULL trong các trigger mức lệnh và cho các hoạt động INSERT.

TG_NAME

> Dạng dữ liệu name; biến có chứa tên của trigger thực sự được thực thi.

TG_WHEN

> Dạng dữ liệu text; một chuỗi hoặc của BEFORE hoặc AFTER phụ thuộc vào định nghĩa trigger.

TG_LEVEL

> Dạng dữ liệu text; một chuỗi của hoặc ROW hoặc STATEMENT phụ thuộc vào định nghĩa trigger.

TG_OP

> Dạng dữ liệu text; một chuỗi của INSERT, UPDATE, DELETE, hoặc TRUNCATE nói vì hoạt động nào trigger đã được thực thi.

TG_RELID

> Dạng dữ liệu oid; ID đối tượng của bảng đã triệu gọi trigger.

TG_RELNAME

> Dạng dữ liệu name; tên của bảng đã triệu gọi trigger. Bây giờ nó bị phản đối, và có thể biến

mất trong một phiên bản trong tương lai. Thay vào đó, hãy sử dụng TG_TABLE_NAME.

TG_TABLE_NAME

Dạng dữ liệu name; tên của bảng đã triệu gọi trigger.

TG_TABLE_SCHEMA

Dạng dữ liệu name; tên của sơ đồ bảng đã triệu gọi trigger.

TG_NARGS

Dạng dữ liệu integer; số các đối số được trao cho thủ tục trigger trong lệnh CREATE TRIGGER.

TG_ARGV[]

Dạng dữ liệu mảng text; các đối số từ lệnh CREATE TRIGGER. Chỉ số tính từ 0. Các chỉ số không hợp lệ (nhỏ hơn 0 hoặc lớn hơn hoặc bằng tg_nargs) sinh ra một giá trị null.

Một hàm trigger phải trả về hoặc NULL hoặc một giá trị bản ghi/hàng có chính xác cấu trúc của bảng mà trigger đó đã được thực thi.

Các trigger mức hàng được thực thi BEFORE có thể trả về null để đánh tín hiệu cho trình quản lý trigger để bỏ qua phần còn lại của hoạt động đối với hàng này (nghĩa là, các trigger tiếp sau sẽ không được thực thi, và INSERT /UPDATE /DELETE không xảy ra cho hàng này). Nếu một giá trị không là null được trả về thì hoạt động đó tiến hành với giá trị hàng đó.

Việc trả về một giá trị hàng khác với giá trị gốc ban đầu của NEW tùy chỉnh hàng sẽ được chèn vào hoặc được cập nhật. Vì thế, nếu hàm trigger muốn làm bật dậy hành động để đi tiếp bình thường mà không có việc tùy biến giá trị hàng đó, thì NEW (hoặc một giá trị bằng ở đó) phải được trả về. Để tùy biến hàng sẽ được lưu trữ, có khả năng để thay thế các giá trị duy nhất trực tiếp trong NEW và trả về NEW được sửa đổi, hoặc để xây dựng một bản ghi/hàng mới hoàn chỉnh để trả về. Trong trường hợp của trigger trước trong DELETE, giá trị được trả về không có hiệu ứng trực tiếp, nhưng nó phải không null để cho phép hành động trigger sẽ tiến hành. Lưu ý rằng NEW là null trong các trigger DELETE, nên việc trả về đó thường không nhạy cảm. Một cách diễn đạt hữu dụng trong các trigger DELETE có lẽ là trả về OLD.

Giá trị trả về của một trigger mức hàng đã thực thi AFTER hoặc một trigger mức lệnh đã thực thi BEFORE hoặc AFTER luôn bị bỏ qua; nó có thể cũng là null. Tuy nhiên, bất kỳ dạng nào của trigger cũng có thể vẫn bỏ qua toàn bộ hoạt động đó bằng việc đưa ra một lỗi.

Ví dụ 39-3 chỉ ra một ví dụ một thủ tục trigger trong PL/pgSQL.

Ví dụ 39-3. Thủ tục trigger PL/pgSQL

Ví dụ trigger này đảm bảo rằng bất kỳ khi nào một hàng được chèn hoặc cập nhật trong bảng, thì tên người sử dụng hiện hành và thời gian được đóng dấu vào trong hàng đó. Và nó kiểm tra xem một tên nhân viên có được đưa ra hay không và lương là một giá trị dương hay không.

```
CREATE TABLE emp (
        empname text,
        salary integer,
        last_date timestamp,
        last_user text
);
```

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
        BEGIN
                -- Check that empname and salary are given
                IF NEW.empname IS NULL THEN
                        RAISE EXCEPTION 'empname cannot be null';
                END IF;
                IF NEW.salary IS NULL THEN
                        RAISE EXCEPTION '% cannot have null salary', NEW.empname;
                END IF;

                -- Who works for us when she must pay for it?
                IF NEW.salary < 0 THEN
                        RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
                END IF;

                -- Remember who changed the payroll when
                NEW.last_date := current_timestamp;
                NEW.last_user := current_user;
                RETURN NEW;
        END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
        FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Một cách khác để lưu ký những thay đổi đối với một bảng liên quan tới việc tạo một bảng mới mà nắm giữ một hàng cho từng lệnh chèn, cập nhật hoặc xóa xảy ra. Tiếp cận này có thể được nghĩ như việc kiểm toán các thay đổi đối với một bảng. Ví dụ 39-4 chỉ ra một ví dụ của một thủ tục trigger kiểm toán trong PL/pgSQL.

Ví dụ 39-4. Thủ tục trigger PL/pgSQL cho việc kiểm toán

Trigger trong ví dụ này đảm bảo rằng bất kỳ lệnh chèn, cập nhật hoặc xóa nào một hàng trong bảng emp cũng được ghi lại (nghĩa là, được kiểm toán) trong bảng emp_audit. Thời gian hiện hành và tên người sử dụng được đóng dấu trong hàng đó, cùng với dạng hoạt động được thực hiện trong đó.

```
CREATE TABLE emp (
        empname text NOT NULL,
        salary integer
);

CREATE TABLE emp_audit(
        operation char(1) NOT NULL,
        stamp timestamp NOT NULL,
        userid text NOT NULL,
        empname text NOT NULL,
        salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
        BEGIN
                --
                -- Create a row in emp_audit to reflect the operation performed on emp,
                -- make use of the special variable TG_OP to work out the operation.
                --
                IF (TG_OP = 'DELETE') THEN
                        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
                        RETURN OLD;
                ELSIF (TG_OP = 'UPDATE') THEN
```

```
                INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
                RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
                INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
                RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Người ta sử dụng các trigger là để duy trì một bảng tóm tắt bảng khác. Kết quả tóm tắt có thể được sử dụng tại chỗ bảng gốc cho các truy vấn nhất định - thường với thời gian chạy được giảm rất nhiều. Kỹ thuật này được sử dụng phổ biến trong Kho Dữ liệu (Data Warehouse), nơi mà các bảng dữ liệu được đo đếm hoặc quan sát (được gọi là các bảng sự việc) có thể cực kỳ lớn. Ví dụ 39-5 chỉ ra ví dụ về một thủ tục trigger trong PL/pgSQL mà duy trì bảng tóm tắt cho một bảng sự việc trong một kho dữ liệu.

Ví dụ 39-5. Thủ tục trigger PL/pgSQL để duy trì một bảng tóm tắt

Sơ đồ được chi tiết hóa ở đây một phần dựa vào ví dụ *Cửa hàng Tạp phẩm* (*Grocery Store*) từ Bộ công cụ *Kho dữ liệu* (*Data Warehouse*) của Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
CREATE TABLE time_dimension (
    time_key integer NOT NULL,
    day_of_week integer NOT NULL,
    day_of_month integer NOT NULL,
    month integer NOT NULL,
    quarter integer NOT NULL,
    year integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key integer NOT NULL,
    product_key integer NOT NULL,
    store_key integer NOT NULL,
    amount_sold numeric(12,2) NOT NULL,
    units_sold integer NOT NULL,
    amount_cost numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key integer NOT NULL,
    amount_sold numeric(15,2) NOT NULL,
    units_sold numeric(12) NOT NULL,
    amount_cost numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);
```

```
--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
        DECLARE
                delta_time_key integer;
                delta_amount_sold numeric(15,2);
                delta_units_sold numeric(12);
                delta_amount_cost numeric(15,2);
        BEGIN

                -- Work out the increment/decrement amount(s).
                IF (TG_OP = 'DELETE') THEN
                        delta_time_key = OLD.time_key;
                        delta_amount_sold = -1 * OLD.amount_sold;
                        delta_units_sold = -1 * OLD.units_sold;
                        delta_amount_cost = -1 * OLD.amount_cost;

                ELSIF (TG_OP = 'UPDATE') THEN

                        -- forbid updates that change the time_key -
                        -- (probably not too onerous, as DELETE + INSERT is how most
                        -- changes will be made).
                        IF ( OLD.time_key != NEW.time_key) THEN
                                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                                OLD.time_key, NEW.time_key;
                        END IF;
                        delta_time_key = OLD.time_key;
                        delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
                        delta_units_sold = NEW.units_sold - OLD.units_sold;
                        delta_amount_cost = NEW.amount_cost – OLD.amount_cost;

                ELSIF (TG_OP = 'INSERT') THEN

                        delta_time_key = NEW.time_key;
                        delta_amount_sold = NEW.amount_sold;
                        delta_units_sold = NEW.units_sold;
                        delta_amount_cost = NEW.amount_cost;

                END IF;

                -- Insert or update the summary row with the new values.
                <<insert_update>>
                LOOP
                        UPDATE sales_summary_bytime
                                SET amount_sold = amount_sold + delta_amount_sold,
                                        units_sold = units_sold + delta_units_sold,
                                        amount_cost = amount_cost + delta_amount_cost
                                WHERE time_key = delta_time_key;

                        EXIT insert_update WHEN found;

                        BEGIN
                                INSERT INTO sales_summary_bytime (
                                                time_key,
                                                amount_sold,
                                                units_sold,
                                                amount_cost)
                                VALUES (
                                                delta_time_key,
```

```
                                        delta_amount_sold,
                                        delta_units_sold,
                                        delta_amount_cost
                                    );
                                EXIT insert_update;

                            EXCEPTION
                                WHEN UNIQUE_VIOLATION THEN
                                -- do nothing
                            END;
                        END LOOP insert_update;

                        RETURN NULL;
                END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
        FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

39.10. Bên trên PL/pgSQL

Phần này mô tả vài chi tiết triển khai là quan trọng thường xuyên đối với những người sử dụng PL/pgSQL để biết.

39.10.1. Thay biến

Các lệnh và các biểu thức SQL trong một hàm PL/pgSQL có thể tham chiếu tới các biến và tham số của hàm. Ở đằng sau, PL/pgSQL thay các tham số truy vấn cho các tham chiếu như vậy. Các tham số sẽ chỉ được thay tại chỗ ở những nơi tham số hoặc tham chiếu cột được phép về mặt cú pháp.

Như một trường hợp ở cực, hãy xem xét ví dụ dạng lập trình tồi này:

```
INSERT INTO foo (foo) VALUES (foo);
```

Trường hợp đầu của foo về cú pháp phải là một tên bảng, nên nó sẽ không được thay thế, thậm chí nếu hàm đó có một biến có tên là foo. Trường hợp thứ 2 phải là tên của một cột của bảng, nên nó cũng sẽ không được thay thế. Chỉ trường hợp 3 là ứng viên sẽ là tham chiếu cho biến của hàm đó.

Lưu ý: Các phiên bản trước 9.0 có thể cố thay biến ở cả 3 trường hợp, dẫn tới lỗi cú pháp.

Vì tên các biến về cú pháp là không khác với tên các cột của bảng, nên có thể có sự tù mù trong các lệnh mà cũng tham chiếu tới các bảng: là một tên được đưa ra có nghĩa để tham chiếu tới cột của một bảng, hoặc một biến? Hãy thay đổi ví dụ trên thành

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Ở đây dest và src phải là các tên bảng, và phải là cột của dest, nhưng foo và bar có lẽ hợp lý hoặc là các biến của hàm hoặc các cột của src.

Mặc định, PL/pgSQL sẽ nêu một lỗi nếu một tên trong một lệnh SQL có thể tham chiếu tới hoặc một biến hoặc một cột của bảng. Bạn có thể sửa vấn đề như vậy bằng việc đổi tên biến hoặc cột, hoặc bằng việc định tính tham chiếu mù mờ đó, hoặc bằng việc nói cho PL/pgSQL sự diễn giải nào sẽ ưu tiên hơn.

Giải pháp đơn giản nhất là đổi tên biến hoặc cột. Một quy tắc lập trình phổ biến là sử dụng qui ước đặt tên khác cho các biến của PL/pgSQL mà bạn sử dụng cho các tên cột. Ví dụ, nếu bạn nhất quán đặt tên cho các biến v_something của hàm trong khi không tên cột nào của bạn bắt đầu bằng v_, thì không xung đột nào xảy ra cả.

Như một sự lựa chọn, bạn có thể định tính các tham chiếu mù mờ để làm cho chúng rõ ràng. Trong ví dụ ở trên, src.foo có thể là một tham chiếu mù mờ đối với cột của bảng. Để tạo một tham chiếu không mù mờ cho một biến, hãy khai báo nó trong một khối có gắn nhãn và sử dụng nhãn của khối đó (xem Phần 39.2). Ví dụ

```
<<block>>
DECLARE
        foo int;
BEGIN
        foo := ...;
        INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Ở đây block.foo có nghĩa là biến thậm chí nếu có cột foo trong src. Các tham số hàm, cũng như các biến đặc biệt như FOUND, có thể được tên hàm định tính, vì chúng được ngầm khai báo cho một khối bên ngoài có gắn nhãn với tên của hàm đó.

Đôi khi là không thực tế để sửa tất cả các tham chiếu mù mờ trong một thân mã lớn của PL/pgSQL. Trong các trường hợp như vậy bạn có thể chỉ định rằng PL/pgSQL sẽ giải quyết các tham chiếu mù mờ như các biến (nó là tương thích với hành vi của PL/pgSQL phiên bản PostgreSQL trước 9.0), hoặc như là cột của bảng (nó là tương thích với vài hệ thống khác như Oracle).

Để thay đổi hành vi này trên cơ sở một hệ thống rộng lớn, hãy thiết lập tham số cấu hình plpgsql.variable_conflict về một trong số error, use_variable, hoặc use_column (trong đó error là yếu tố mặc định). Tham số này ảnh hưởng tới các biên dịch các lệnh sau đó trong các hàm của PL/pgSQL, nhưng không với các lệnh được biên dịch rồi trong phiên hiện hành. Để thiết lập tham số đó trước khi PL/pgSQL được tải lên, là cần thiết đã thêm "plpgsql" vào danh sách custom_variable_classes trong postgresql.conf. Vì việc thay đổi thiết lập này có thể gây ra những thay đổi không mong muốn trong hành vi của các hàm PL/pgSQL, nó chỉ có thể được một siêu người sử dụng thay đổi.

Bạn cũng có thể thiết lập hành vi đó trên cơ sở từng hàm một, bằng việc chèn một trong các lệnh đặc biệt đó ở đầu của văn bản hàm đó:

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

Các lệnh đó chỉ ảnh hưởng tới hàm mà chúng được viết, và viết đè lên thiết lập của plpgsql.variable_conflict. Một ví dụ là

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
        #variable_conflict use_variable
        DECLARE
                curtime timestamp := now();
        BEGIN
                UPDATE users SET last_modified = curtime, comment = comment
                 WHERE users.id = id;
        END;
$$ LANGUAGE plpgsql;
```

Trong lệnh UPDATE, curtime, comment, và id sẽ tham chiếu tới các biến và tham số của hàm đó bất kể có hay không có các cột users các tên đó. Lưu ý là chúng ta đã phải định tính tham chiếu tới users.id trong mệnh đề WHERE để làm cho nó tham chiếu tới cột của bảng đó. Nhưng chúng ta đã không phải định tính tham chiếu tới comment như một cái đích trong danh sách UPDATE, vì về cú pháp điều đó phải là một cột users. Chúng ta có thể viết hàm y hệt mà không phụ thuộc vào thiết lập variable_conflict theo cách này:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
        <<fn>>
        DECLARE
                curtime timestamp := now();
        BEGIN
                UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
                 WHERE users.id = stamp_user.id;
        END;
$$ LANGUAGE plpgsql;
```

Sự thay biến không xảy ra trong chuỗi lệnh được đưa ra cho EXECUTE hoặc một trong các phương án của nó. Nếu bạn cần chèn giá trị khác nhau vào một lệnh như vậy, hãy làm thế như một phần của việc xây dựng giá trị chuỗi, hoặc hãy sử dụng USING, như được minh họa trong Phần 39.5.4.

Thay biến hiện hành chỉ làm việc trong các lệnh SELECT, INSERT, UPDATE, và DELETE, vì máy SQL chính cho phép các tham số truy vấn chỉ trong các lệnh đó. Để sử dụng một tên hoặc giá trị không phải là hằng trong các dạng lệnh khác (thường được gọi là các lệnh tiện ích), thì bạn phải xây dựng lệnh tiện ích như một chuỗi và EXECUTE nó.

39.10.2. Lưu giữ kế hoạch

Trình biên dịch của PL/pgSQL phân tích cú pháp văn bản nguồn của hàm và lần đầu tiên tạo ra một cây chỉ lệnh nhị phân bên trong hàm được gọi (trong từng phiên làm việc). Cây chỉ lệnh đó dịch đầy đủ cấu trúc lệnh PL/pgSQL, nhưng các biểu thức SQL và các lệnh SQL riêng rẽ được sử dụng trong hàm không được dịch ngay lập tức.

Khi từng biểu thức và lệnh SQL lần đầu được thực thi trong hàm, trình biên dịch của PL/pgSQL tạo kế hoạch thực thi chuẩn bị sẵn (sử dụng các hàm SPI_prepare và SPI_saveplan của trình quản lý SPI). Các cuộc viếng thăm sau đó tới biểu thức hoặc lệnh đó sử dụng kế hoạch được chuẩn bị sẵn đó. Vì thế một hàm với mã điều kiện chứa nhiều lệnh theo đó các kế hoạch thực thi có thể được yêu cầu sẽ chỉ chuẩn bị và lưu các kế hoạch nào thực sự sẽ được sử dụng trong vòng đời kết nối cơ sở dữ liệu.

Điều này có thể làm giảm đáng kể tổng số lượng thời gian được yêu cầu để phân tích cú pháp và sinh ra các kế hoạch thực thi cho các lệnh trong một hàm của PL/pgSQL. Một nhược điểm là các lỗi trong một biểu thức hoặc lệnh đặc thù không thể được dò tìm ra cho tới khi phần đó của hàm được với tới trong thực thi. (Các lỗi cú pháp tầm thường sẽ được dò tìm ra trong quá trình truyền ban đầu phân tích cú pháp, nhưng bất kỳ điều gì sâu hơn sẽ không được dò tìm ra cho tới khi thực thi).

Một kế hoạch được lưu giữ sẽ tự động được lên lại kế hoạch nếu có bất kỳ sự thay đổi sơ đồ nào cho bất kỳ bảng nào được sử dụng trong truy vấn đó, hoặc nếu bất kỳ hàm nào do người sử dụng định nghĩa được sử dụng trong truy vấn được định nghĩa lại. Điều này làm cho sự sử dụng lại các kế hoạch được chuẩn bị sẵn là minh bạch trong hầu hết các trường hợp, mà có các trường hợp đặc biệt nơi mà một kế hoạch giai đoạn nào đó có thể được sử dụng lại. Một ví dụ là việc bỏ và tạo lại một toán tử do người sử dụng định nghĩa sẽ không ảnh hưởng tới các kế hoạch được lưu giữ rồi; chúng sẽ tiếp tục gọi hàm bên dưới của toán tử gốc ban đầu, nếu điều đó còn chưa được thay đổi. Khi cần, bộ nhớ trữ tạm (cache) có thể được xả bằng việc bắt đầu một phiên làm việc mới của cơ sở dữ liệu.

Vì PL/pgSQL lưu các kế hoạch thực thi theo cách này, các lệnh SQL mà xuất hiện trực tiếp trong một hàm PL/pgSQL phải tham chiếu tới các bảng và các cột y hệt trong từng thực thi; đó là, bạn không thể sử dụng một tham số như là tên của một bảng hoặc cột trong một lệnh SQL. Để có hạn chế này, bạn có thể xây dựng các lệnh động bằng việc sử dụng lệnh EXECUTE của PL/pgSQL - với cái giá của việc xây dựng một kế hoạch thực thi mới trong từng sự thực thi.

Một điểm quan trọng khác là các kế hoạch được chuẩn bị sẽ được tham số hóa để cho phép các giá trị các biến của PL/pgSQL thay đổi từ sự sử dụng này sang sự sử dụng tiếp theo, như được thảo luận chi tiết ở trên. Đôi khi điều này có nghĩa là một kế hoạch là ít hiệu quả hơn là nó có thể nếu được sinh ra cho một giá trị biến đặc biệt. Như một ví dụ, hãy xem xét

```
SELECT * INTO myrec FROM dictionary WHERE word LIKE search_term;
```

trong đó search_term là một biến của PL/pgSQL. Kế hoạch được lưu giữ cho truy vấn này sẽ không bao giờ sử dụng một chỉ số trong word, vì trình lên kế hoạch không thể giả thiết rằng mẫu LIKE sẽ được định vị trái vào lúc chạy. Để sử dụng một chỉ số thì truy vấn phải được lên kế hoạch với một mẫu hằng số LIKE được cung cấp. Đây là tình huống khác nơi mà EXECUTE có thể được sử dụng để ép một kế hoạch mới sẽ được sinh ra cho từng sự thực thi.

Bản chất tự nhiên hay thay đổi của các biến bản ghi thể hiện một vấn đề khác trong kết nối này. Khi các trường của một biến bản ghi được sử dụng trong các biểu thức hoặc các lệnh, các dạng dữ liệu của các trường đó phải không thay đổi từ lời gọi hàm này sang lời gọi hàm tiếp theo vì từng biểu thức sẽ được lên kế hoạch bằng việc sử dụng dạng dữ liệu đang hiện diện khi biểu thức đó lần đầu tiên được với tới. EXECUTE có thể được sử dụng để khắc phục vấn đề này khi cần thiết.

Nếu hàm y hệt được sử dụng như một trigger cho hơn 1 bảng, thì PL/pgSQL chuẩn bị và lưu giữ kế hoạch ngay lập tức cho từng bảng như vậy - đó là, có bộ nhớ tạm cho từng hàm trigger và sự kết hợp bảng, chứ không chỉ cho từng hàm. Điều này làm dịu bớt một số vấn đề với các dạng dữ liệu khác nhau; ví dụ, một hàm trigger sẽ có khả năng làm việc thành công với một cột có tên key thậm chí nếu nó bỗng nhiên có các dạng khác nhau trong các bảng khác nhau.

Tương tự, các hàm có các dạng đối số đa hình có một bộ nhớ tạm cho kế hoạch riêng rẽ cho từng sự kết hợp các dạng đối số thực tế mà chúng từng được triệu gọi, sao cho những khác biệt các dạng dữ liệu đó không gây ra những hỏng hóc không ngờ tới.

Việc lưu giữ kế hoạch đôi khi có thể có các hiệu ứng đáng ngạc nhiên trong sự diễn giải các giá trị nhạy cảm về thời gian. Ví dụ có một sự khác biệt giữa những gì 2 hàm sau làm:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
        BEGIN
                INSERT INTO logtable VALUES (logtxt, 'now');
        END;
$$ LANGUAGE plpgsql;
```

và:

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
        DECLARE
                curtime timestamp;
        BEGIN
                curtime := 'now';
                INSERT INTO logtable VALUES (logtxt, curtime);
        END;
$$ LANGUAGE plpgsql;
```

Trong trường hợp của logfunc1, trình phân tích cú pháp chính của PostgreSQL biết khi chuẩn bị kế hoạch cho INSERT mà chuỗi 'now' sẽ được dịch như là timestamp, vì cột đích của logtable là ở dạng đó. Vì 'now' sẽ được chuyển đổi thành một hằng khi INSERT được lên kế hoạch, và sau đó được sử dụng trong tất cả các triệu gọi của logfunc1 trong vòng đời của phiên làm việc. Cần phải nói rằng, đây không phải những gì một lập trình viên muốn.

Trong trường hợp của logfunc2, trình phân tích cú pháp chính của PostgreSQL không biết dạng nào 'now' sẽ trở thành và vì thế nó trả về một giá trị dữ liệu dạng text chứa chuỗi now. Trong quá trình đảm bảo chỉ định tới biến cục bộ curtime, trình biên dịch của PL/pgSQL đưa ra chuỗi này tới dạng timestamp bằng việc gọi các hàm text_out và timestamp_in để chuyển đổi. Vì thế, dấu thời gian được tính toán sẽ được cập nhật mỗi lần thực thi như lập trình viên kỳ vọng.

39.11. Mẹo cho việc phát triển trong PL/pgSQL

Một cách tốt để phát triển trong PL/pgSQL là sử dụng trình soạn thảo văn bản theo lựa chọn của bạn để tạo các hàm của bạn, và trong một cửa sổ khác, hãy sử dụng psql để tải và kiểm thử các hàm đó. Nếu bạn đang thực hiện điều đó theo cách này, là ý tưởng tốt để viết hàm đó bằng việc sử dụng CREATE OR REPLACE FUNCTION. Cách đó bạn có thể chỉ tải lại tệp để cập nhật định nghĩa hàm. Ví dụ:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
        ....
$$ LANGUAGE plpgsql;
```

Trong khi chạy psql, bạn có thể tải hoặc tải lại một tệp định nghĩa hàm như vậy với:

```
\i filename.sql
```

và sau đó ngay lập tức đưa ra các lệnh SQL để kiểm thử hàm đó.

Một cách tốt nữa để phát triển trong PL/pgSQL là với một công cụ truy cập cơ sở dữ liệu GUI (giao diện đồ họa người sử dụng) mà tạo thuận lợi cho sự phát triển trong một ngôn ngữ thủ tục. Một ví dụ về một công cụ như vậy là pgAdmin, dù các công cụ khác cũng tồn tại. Các công cụ đó thường cung cấp các tính năng thuận tiện như việc thoát ra các dấu ngoặc đơn và làm cho dễ dàng hơn để tái tạo và gỡ lỗi các hàm.

39.11.1. Điều khiển dấu trích dẫn

Mã của một hàm PL/pgSQL được chỉ định trong CREATE FUNCTION như một hằng chuỗi. Nếu bạn viết hằng chuỗi đó theo cách thông thường với các dấu ngoặc đơn xung quanh, thì bất kỳ dấu ngoặc đơn nào bên trong thân hàm cũng phải được đúp bản; tương tự bất kỳ dấu chéo ngược nào cũng phải được đúp bản (giả thiết cú pháp chuỗi thoát được sử dụng). Việc đúp bản các dấu ngoặc là nặng nề, và trong các trường hợp phức tạp hơn thì mã có thể trở thành thực sự không thể hiểu, vì bạn có thể dễ dàng tự bạn thấy cần nửa tá hoặc hơn các dấu ngoặc liền kề nhau. Được khuyến cáo là bạn thay vào đó hãy viết thân hàm như một hằng chuỗi "nằm trong các dấu $" (xem Phần 4.1.2.4).

Theo tiếp cận các dấu $, bạn không bao giờ đúp bản bất kỳ dấu ngoặc nào, mà thay vào đó chăm sóc để chọn một dấu phân cách các dấu $ khác nhau cho từng mức lồng bạn cần. Ví dụ, bạn có lẽ viết lệnh CREATE FUNCTION như:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE plpgsql;
```

Trong việc này, bạn có thể sử dụng các dấu ngoặc cho các chuỗi hằng đơn giản trong các lệnh SQL và các dấu $$ để phân cách các phân đoạn các lệnh SQL mà bạn lắp thành các chuỗi. Nếu bạn cần văn bản trích dẫn bao gồm các các dấu $$, thì bạn có thể sử dụng Q, và cứ như thế.

Phần sau đây chỉ ra những gì bạn phải làm khi viết các dấu ngoặc mà không có các dấu $. Có lẽ là hữu dụng khi dịch mã việc trích dẫn bằng các dấu $ thành thứ gì đó toàn diện hơn.

1 dấu trích dẫn

Để bắt đầu và kết thúc thân hàm, ví dụ:

```
CREATE FUNCTION foo() RETURNS integer AS '
        ....
' LANGUAGE plpgsql;
```

Bất kỳ ở đâu trong thân hàm các dấu trích dẫn

2 dấu trích dẫn

Đối với các hằng chuỗi bên trong thân hàm, ví dụ:

```
a_output := "Blah";
SELECT * FROM users WHERE f_name="foobar";
```

Theo tiếp cận trích dẫn bằng dấu $, bạn chỉ viết:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

nó là chính xác những gì trình phân tích cú pháp PL/pgSQL có thể thấy ở cả 2 trường hợp.

4 dấu trích dẫn

Khi bạn cần một dấu trích dẫn đơn trong một hằng chuỗi bên trong thân hàm, ví dụ:

```
a_output := a_output || ” AND name LIKE ””foobar”” AND xyz”
```

Giá trị thực sự được nối tới a_output có thể là: AND name LIKE 'foobar' AND xyz.

Theo tiếp cận dấu $, bạn có thể viết:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

hãy cẩn thận là bất kỳ sự phân cách nào bằng dấu $ xung quanh điều này không chỉ là $$.

6 dấu trích dẫn

Khi một dấu trích dẫn đơn trong một chuỗi bên trong thân hàm là liền kề với kết thúc của hằng chuỗi đó, ví dụ:

```
a_output := a_output || ” AND name LIKE ””foobar”””
```

Giá trị được nói tới a_output có thể sau đó là: AND name LIKE 'foobar'.

Theo tiếp cận dấu $, điều này trở thành:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 dấu trích dẫn

Khi bạn muốn 2 dấu trích dẫn đơn trong một hằng chuỗi (nó tính cho 8 dấu trích dẫn) và điều này là liền kề với cuối của hằng chuỗi đó (2 hơn). Bạn sẽ có thể chỉ cần điều đó nếu bạn đang viết một hàm mà sinh ra các hàm khác, như trong Ví dụ 39-7. Ví dụ:

```
a_output := a_output || ” if v_” ||
        referrer_keys.kind || ” like ”””””
        || referrer_keys.key_string || ”””””
        then return ””” || referrer_keys.referrer_type
        || ”””; end if;”;
```

Giá trị của a_output có thể sau đó là:

```
if v_... like ”...” then return ”...”; end if;
```

Theo tiếp cận dấu $, điều này trở thành:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
        || referrer_keys.key_string || $$'
        then return '$$ || referrer_keys.referrer_type
        || $$'; end if;$$;
```

trong đó chúng ta giả thiết chúng ta chỉ cần đặt các dấu nháy đơn (') vào a_output, vì nó sẽ được tái trích dẫn trước khi sử dụng.

39.12. Chuyển từ Oracle PL/SQL

Phần này giải thích những khác biệt giữa ngôn ngữ PL/pgSQL của PostgreSQL và ngôn ngữ PL/pgSQL của Oracle, để giúp các lập trình viên muốn chuyển các ứng dụng từ Oracle sang PostgreSQL.

PL/pgSQL là tương tự với PL/SQL theo nhiều khía cạnh. Đây là một ngôn ngữ lệnh, cấu trúc khối, và tất cả các biến phải được khai báo. Các chỉ định, vòng lặp, điều kiện là tương tự. Các khác biệt chính bạn nên giữ trong đầu khi chuyển từ PL/SQL sang PL/pgSQL là:

- Nếu một tên được sử dụng trong một lệnh SQL có thể hoặc là một tên cột của một bảng hoặc là một tham chiếu tới một biến của hàm, thì PL/SQL ứng xử với nó như một tên cột. Điều này tương ứng với hành vi plpgsql.variable_conflict = use_column của PL/gpSQL, nó không là mặc định, như được giải thích trong Phần 39.10.1. Thường là tốt nhất để tránh những mù mờ như vậy ngay từ đầu, nhưng nếu bạn phải chuyển một lượng mã lớn mà phụ thuộc vào hành vi này, thì việc thiết lập variable_conflict có thể là giải pháp tốt nhất.
- Trong PostgreSQL thân hàm phải được viết như một hằng chuỗi. Vì thế bạn cần phải sử dụng các dấu $ hoặc các dấu ngoặc đơn thoát trong thân hàm. (Xem Phần 39.11.1)
- Thay vì các gói, hãy sử dụng các sơ đồ để tổ chức các hàm của bạn trong các nhóm.
- Vì không có các gói, nên sẽ không có các biến mức gói. Điều này hơi phiền phức. Thay vào đó, bạn có thể giữ tình trạng theo từng phiên trong các bảng tạm.
- Các vòng lặp số nguyên FOR với REVERSE làm việc khác nhau: PL/SQL tính xuống từ số thứ 2 tới số thứ nhất, trong khi PL/pgSQL tính xuống từ số thứ nhất tới số thứ 2, đòi hỏi các ràng buộc vòng lặp sẽ được hoán đổi khi chuyển. Tính không tương thích này là không may nhưng không có khả năng sẽ được thay đổi. (Xem Phần 39.6.3.5).
- Các vòng lặp FOR đối với các truy vấn (khác với các con trỏ) cũng làm việc khác: (các) biến đích phải được khai báo, trong khi PL/SQL luôn khai báo ngầm chúng. Một ưu điểm của điều này là các giá trị biến vẫn còn truy cập được sau khi thoát vòng lặp.
- Có những khác biệt về ký hiệu khác nhau để sử dụng các biến con trỏ.

39.12.1. Ví dụ chuyển

Ví dụ 39-6 chỉ cách chuyển một hàm từ PL/SQL sang PL/pgSQL.

Ví dụ 39-6. Chuyển một hàm đơn giản từ PL/SQL sang PL/pgSQL

Đây là một hàm PL/SQL của Oracle:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURN varchar IS
BEGIN
        IF v_version IS NULL THEN
                RETURN v_name;
        END IF;
        RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Hãy đi qua hàm này và thấy những khác biệt so với PL/pgSQL:

- Từ khóa RETURN trong nguyên mẫu hàm (không phải thân hàm) thành RETURNS trong PostgreSQL. Hơn nữa IS thành AS, và bạn cần thêm một mệnh đề LANGUAGE vì PL/pgSQL là không chỉ có thể là ngôn ngữ hàm.
- Trong PostgreSQL, thân hàm được coi là một hằng chuỗi, nên bạn cần sử dụng các dấu ngoặc kép hoặc các dấu $ xung quanh nó. Điều này thay thế cho dấu kết thúc / trong tiếp cận

của Oracle.

- Lệnh show errors không tồn tại trong PostgreSQL, và là không cần thiết vì các lỗi được nêu tự động.

Đây là cách mà hàm này có thể trông giống khi được chuyển sang PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
        IF v_version IS NULL THEN
                RETURN v_name;
        END IF;
        RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Ví dụ 39-7 chỉ cách để chuyển một hàm tạo ra hàm khác và cách để điều khiển các vấn đề dấu ngoặc tiếp theo.

Ví dụ 39-7. Chuyển một hàm mà tạo ra hàm khác từ PL/SQL sang PostgreSQL

Thủ tục sau chộp các hàng từ một lệnh SELECT và xây dựng một hàm lớn với các kết quả trong các lệnh IF, vì sự có lợi cho tính hiệu quả.

Đây là phiên bản của Oracle:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
        CURSOR referrer_keys IS
                SELECT * FROM cs_referrer_keys
                ORDER BY try_order;
        func_cmd VARCHAR(4000);
BEGIN
        func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
                v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

        FOR referrer_key IN referrer_keys LOOP
             func_cmd := func_cmd ||
                ' IF v_' || referrer_key.kind
                || ' LIKE "' || referrer_key.key_string
                || '" THEN RETURN "' || referrer_key.referrer_type
                || '"; END IF;';
        END LOOP;
        func_cmd := func_cmd || ' RETURN NULL; END;';

        EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Đây là cách mà hàm này có thể kết thúc trong PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
        referrer_keys CURSOR IS
                SELECT * FROM cs_referrer_keys
                ORDER BY try_order;
        func_body text;
        func_cmd text;
BEGIN
```

```
        func_body := 'BEGIN';

        FOR referrer_key IN referrer_keys LOOP
            func_body := func_body ||
                ' IF v_' || referrer_key.kind
                || ' LIKE ' || quote_literal(referrer_key.key_string)
                || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
                || '; END IF;' ;
        END LOOP;

        func_body := func_body || ' RETURN NULL; END;';

        func_cmd :=
                'CREATE OR REPLACE FUNCTION cs_find_referrer_type (v_host varchar,
                                                                   v_domain varchar,
                                                                   v_url varchar)
                RETURNS varchar AS '
                || quote_literal(func_body)
                || ' LANGUAGE plpgsql;' ;

        EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Lưu ý cách mà thân hàm được xây dựng tách bạch và được truyền qua quote_literal tới bất kỳ dấu ngoặc kép đúp bản nào trong nó. Kỹ thuật này là cần thiết vì chúng ta không thể sử dụng an toàn các dấu $ để xác định hàm mới: chúng ta không biết chắc các chuỗi nào sẽ được nội suy từ trường referrer_key.key_string. (Chúng ta đang giả thiết ở đây rằng referrer_key.kind có thể được tin cậy luôn là host, domain, hoặc url, nhưng referrer_key.key_string có thể là bất kỳ điều gì, đặc biệt nó có thể chứa các dấu $). Hàm này thực sự là một cải tiến trong gốc Oracle, vì nó sẽ không sinh ra mã đứt đoạn khi referrer_key.key_string hoặc referrer_key.referrer_type có chứa các dấu ngoặc kép.

Ví dụ 39-8 chỉ cách chuyển một hàm với các tham số và điều khiển chuỗi OUT. PostgreSQL không có một hàm được xây dựng sẵn, nhưng bạn có thể tạo hàm bằng việc sử dụng một sự kết hợp các hàm khác. Trong Phần 39.12.3 có một triển khai PL/pgSQL của instr mà bạn có thể sử dụng để làm cho việc chuyển của bạn dễ dàng hơn.

Ví dụ 39-8. Chuyển một thủ tục với điều khiển chuỗi và các tham số OUT từ PL/SQL sang PL/pgSQL

Thủ tục sau đây của Oracle PL/SQL được sử dụng để phân tích cú pháp một URL và trả về vài phần tử (máy chủ host, đường dẫn, và truy vấn):

Đây là phiên bản Oracle:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
        v_url IN VARCHAR,
        v_host OUT VARCHAR, -- This will be passed back
        v_path OUT VARCHAR, -- This one too
        v_query OUT VARCHAR) -- And this one
IS
        a_pos1 INTEGER;
        a_pos2 INTEGER;
BEGIN
        v_host := NULL;
        v_path := NULL;
        v_query := NULL;
```

```
        a_pos1 := instr(v_url, '//');

        IF a_pos1 = 0 THEN
                RETURN;
        END IF;
        a_pos2 := instr(v_url, '/', a_pos1 + 2);
        IF a_pos2 = 0 THEN
                v_host := substr(v_url, a_pos1 + 2);
                v_path := '/';
                RETURN;
        END IF;

        v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
        a_pos1 := instr(v_url, '?', a_pos2 + 1);

        IF a_pos1 = 0 THEN
                v_path := substr(v_url, a_pos2);
                RETURN;
        END IF;

        v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
        v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Đây là một bản dịch có khả năng trong PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
        v_url IN VARCHAR,
        v_host OUT VARCHAR, -- This will be passed back
        v_path OUT VARCHAR, -- This one too
        v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
        a_pos1 INTEGER;
        a_pos2 INTEGER;
BEGIN
        v_host := NULL;
        v_path := NULL;
        v_query := NULL;
        a_pos1 := instr(v_url, '//');

        IF a_pos1 = 0 THEN
                RETURN;
        END IF;
        a_pos2 := instr(v_url, '/', a_pos1 + 2);
        IF a_pos2 = 0 THEN
                v_host := substr(v_url, a_pos1 + 2);
                v_path := '/';
                RETURN;
        END IF;

        v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
        a_pos1 := instr(v_url, '?', a_pos2 + 1);

        IF a_pos1 = 0 THEN
                v_path := substr(v_url, a_pos2);
                RETURN;
        END IF;
        v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
        v_query := substr(v_url, a_pos1 + 1);
```

```
END;
$$ LANGUAGE plpgsql;
```

Hàm này có thể được sử dụng như thế này:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Ví dụ 39-9 chỉ cách chuyển một thủ tục sử dụng nhiều tính năng là đặc thù với Oracle.

Ví dụ 39-9. Chuyển một thủ tục từ PL/SQL sang PL/pgSQL

Phiên bản Oracle:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
        a_running_job_count INTEGER;
        PRAGMA AUTONOMOUS_TRANSACTION;[1]
BEGIN
        LOCK TABLE cs_jobs IN EXCLUSIVE MODE;[2]

        SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

        IF a_running_job_count > 0 THEN
                COMMIT; -- free lock[3]
                raise_application_error(-20000,
                        'Unable to create a new job: a job is currently running.');
        END IF;

        DELETE FROM cs_active_job;
        INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

        BEGIN
                INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
        EXCEPTION
                WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
        END;
        COMMIT;
END;
/
show errors
```

Các thủ tục như thế này có thể dễ dàng được chuyển đổi thành các hàm PostgreSQL trả về void. Thủ tục này đặc biệt thú vị vì nó có thể dạy chúng ta vài điều:

[1] Không có lệnh PRAGMA trong PostgreSQL.

[2] Nếu bạn làm một LOCK TABLE trong PL/pgSQL, thì khóa đó sẽ không được phát hành cho tới khi việc gọi giao dịch được kết thúc.

[3] Bạn không thể đưa ra COMMIT trong một hàm PostgreSQL. Hàm đó đang chạy bên trong vài giao dịch bên ngoài và vì thế COMMIT có thể ngụ ý việc kết thúc thực thi hàm đó. Tuy nhiên, trong trường hợp đặc biệt này, không nhất thiết bất kỳ cách gì, vì sự khóa mà LOCK TABLE giành được sẽ được phát hành khi chúng ta đưa ra một lỗi.

Đây là cách mà chúng ta có thể chuyển thủ tục này sang PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
        a_running_job_count integer;
BEGIN
```

```
        LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

        SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

        IF a_running_job_count > 0 THEN
                RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; [1]
        END IF;

        DELETE FROM cs_active_job;
        INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

        BEGIN
                INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
        EXCEPTION
                WHEN unique_violation THEN [2]
        -- don't worry if it already exists
        END;
END;
$$ LANGUAGE plpgsql;
```

[1] Cú pháp của RAISE là khác đáng kể với lệnh của Oracle, dù trường hợp cơ bản RAISE exception_name làm việc tương tự.

[2] Các tên ngoại lệ được PL/pgSQL hỗ trợ là khác với của Oracle. Tập hợp các tên ngoại lệ được xây dựng sẵn là lớn hơn nhiều (xem Phụ lục A). Hiện không có cách để khai báo các tên ngoại lệ do người sử dụng khai báo, dù bạn có thể đưa ra các giá trị SQLSTATE do người sử dụng chọn. Sự khác biệt về chức năng chính giữa thủ tục này và thứ tương đương của Oracle là thứ khóa độc quyền trong bảng cs_jobs sẽ được giữ cho tới khi việc gọi giao dịch kết thúc. Hơn nữa, nếu trình gọi sau đó bỏ qua (ví dụ vì một lỗi), thì các ảnh hưởng của thủ tục này sẽ được lăn ngược về như cũ.

39.12.2. Điều khác để xem

Phần này giải thích ít điều khác để xem khi chuyển các hàm PL/SQL của Oracle sang PostgreSQL.

39.12.2.1 Ngầm quay ngược về sau các ngoại lệ

Trong PostgreSQL, khi một biểu thức được một mệnh đề EXCEPTION bắt được, thì tất cả những thay đổi dữ liệu kể từ BEGIN của khối sẽ tự động quay ngược lại lúc trước. Hành vi đó là tương đương với những gì bạn có trong Oracle với:

```
BEGIN
        SAVEPOINT s1;
        ... code here ...
EXCEPTION
        WHEN ... THEN
                ROLLBACK TO s1;
                ... code here ...
        WHEN ... THEN
                ROLLBACK TO s1;
                ... code here ...
END;
```

Nếu bạn đang dịch một thủ tục của Oracle mà sử dụng SAVEPOINT và trong dạng này, thì nhiệm vụ của bạn là dễ dàng: chỉ bỏ qua SAVEPOINT và ROLLBACK TO. Nếu bạn có một thủ tục mà sử dụng

SAVEPOINT và ROLLBACK TO theo một cách khác thì vài suy nghĩ thực tế sẽ được yêu cầu.

39.12.2.2. EXECUTE

Phiên bản PL/pgSQL của EXECUTE làm việc tương tự như phiên bản PL/SQL, nhưng bạn phải nhớ sử dụng quote_literal và quote_ident như được mô tả trong Phần 39.5.4. Các cấu trúc dạng EXECUTE 'SELECT * FROM $1'; sẽ không làm việc tin cậy trừ phi bạn sử dụng các hàm đó.

39.12.2.3. Tối ưu hóa các hàm PL/pgSQL

PostgreSQL trao cho bạn 2 trình sửa đổi tạo hàm để tối ưu hóa sự thực thi: "tính không kiên định" (liệu hàm có luôn trả về kết quả y hệt khi trao cho các đối số y hệt hay không) và "tính khắt khe" (liệu hàm có trả về null nếu bất kỳ đối số nào là null hay không). Hãy tư vấn trang tham chiếu CREATE FUNCTION để có các chi tiết.

Khi sử dụng các thuộc tính tối ưu hóa đó, lệnh CREATE FUNCTION của bạn có lẽ trông giống thế này:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

39.12.3. Phụ lục

Phần này có mã cho một tập hợp các hàm instr tương thích với Oracle mà bạn có thể sử dụng để đơn giản hóa các nỗ lực chuyển của bạn.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2, [n], [m]) where [] denotes optional parameters.
--
-- Searches string1 beginning at the nth character for the mth occurrence
-- of string2. If n is negative, search backwards. If m is not passed,
-- assume 1 (search starts at first character).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
        pos integer;
BEGIN
        pos:= instr($1, $2, 1);
        RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
        pos integer NOT NULL DEFAULT 0;
        temp_str varchar;
        beg integer;
        length integer;
        ss_length integer;
BEGIN
        IF beg_index > 0 THEN
                temp_str := substring(string FROM beg_index);
                pos := position(string_to_search IN temp_str);

                IF pos = 0 THEN
                        RETURN 0;
                ELSE
```

```
                        RETURN pos + beg_index - 1;
                END IF;
        ELSE
                ss_length := char_length(string_to_search);
                length := char_length(string);
                beg := length + beg_index - ss_length + 2;

                WHILE beg > 0 LOOP
                        temp_str := substring(string FROM beg FOR ss_length);
                        pos := position(string_to_search IN temp_str);

                        IF pos > 0 THEN
                                RETURN beg;
                        END IF;

                        beg := beg – 1;
                END LOOP;
                RETURN 0;
        END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar,
                        beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
        pos integer NOT NULL DEFAULT 0;
        occur_number integer NOT NULL DEFAULT 0;
        temp_str varchar;
        beg integer;
        i integer;
        length integer;
        ss_length integer;
BEGIN
        IF beg_index > 0 THEN
                beg := beg_index;
                temp_str := substring(string FROM beg_index);

                FOR i IN 1..occur_index LOOP
                        pos := position(string_to_search IN temp_str);

                        IF i = 1 THEN
                                beg := beg + pos - 1;
                        ELSE
                                beg := beg + pos;
                        END IF;

                        temp_str := substring(string FROM beg + 1);
                END LOOP;

                IF pos = 0 THEN
                        RETURN 0;
                ELSE
                        RETURN beg;
                END IF;
        ELSE
                ss_length := char_length(string_to_search);
                length := char_length(string);
                beg := length + beg_index - ss_length + 2;

                WHILE beg > 0 LOOP
                        temp_str := substring(string FROM beg FOR ss_length);
```

```
                        pos := position(string_to_search IN temp_str);

                IF pos > 0 THEN
                        occur_number := occur_number + 1;

                        IF occur_number = occur_index THEN
                                RETURN beg;
                        END IF;
                END IF;

                beg := beg – 1;

                END LOOP;

                RETURN 0;
        END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chương 40. PL/Tcl - Ngôn ngữ thủ tục Tcl

PL/Tcl là một ngôn ngữ thủ tục cho hệ thống cơ sở dữ liệu PostgreSQL cho phép ngôn ngữ Tcl[1] sử dụng được để viết các hàm và các thủ tục trigger.

40.1. Tổng quan

PL/Tcl đưa ra hầu hết các khả năng một hàm viết có trong ngôn ngữ C, với ít hạn chế, và với sự bổ sung thêm các thư viện xử lý chuỗi mạnh mà có sẵn trong Tcl.

Một giới hạn *tốt* khá lôi cuốn là mọi điều được thực thi từ bên trong sự an toàn của ngữ cảnh trình biên dịch Tcl. Hơn nữa tập hợp lệnh có giới hạn của Tcl an toàn, chỉ một ít lệnh là sẵn sàng để truy cập cơ sở dữ liệu thông qua SPI và để đưa ra các thông điệp qua elog(). PL/Tcl không đưa ra cách thức để truy cập các phần nội bộ của máy chủ cơ sở dữ liệu hoặc giành được sự truy cập mức hệ điều hành (OS) với sự cho phép của tiến trình máy chủ PostgreSQL, như một hàm C có thể làm. Vì thế, người sử dụng cơ sở dữ liệu không có quyền ưu tiên có thể được tin cậy để sử dụng ngôn ngữ này; nó không trao cho họ ủy quyền không giới hạn.

Giới hạn triển khai nổi bật khác là các hàm Tcl không thể được sử dụng để tạo các hàm đầu vào/đầu ra cho các dạng dữ liệu mới.

Đôi khi có mong muốn viết các hàm Tcl mà không bị hạn chế tới Tcl an toàn. Ví dụ, người ta có thể muốn một hàm Tcl gửi thư điện tử. Để điều khiển các trường hợp này, có một phương án PL/Tcl gọi PL/TclU (viết tắt chữ tiếng Anh của không tin cậy [Untrusted]). Điều này chính xác là ngôn ngữ y hệt ngoại trừ một trình biên dịch Tcl đầy đủ được sử dụng. *Nếu PL/TclU được sử dụng, nó phải được cài đặt như một ngôn ngữ thủ tục không tin cậy* sao cho chỉ các siêu người sử dụng cơ sở dữ liệu có thể tạo được các hàm trong nó. Người viết một hàm PL/TclU phải chú ý rằng hàm đó không thể được sử dụng để làm bất kỳ điều gì không mong muốn, vì nó sẽ có khả năng để làm bất kỳ điều gì có thể được một người sử dụng đã đăng nhập như một quản trị hệ thống cơ sở dữ liệu làm.

Mã đối tượng được chia sẻ cho các trình điều khiển lời gọi của PL/Tcl và PL/TclU tự động được xây dựng và được cài đặt trong thư mục thư viện PostgreSQL nếu sự hỗ trợ Tcl được chỉ định trong bước cấu hình của thủ tục cài đặt. Để cài đặt PL/Tcl và/hoặc PL/TclU trong một cơ sở dữ liệu đặc biệt, hãy sử dụng chương trình createlang, ví dụ createlang pltcl *dbname* hoặc createlang pltclu *dbname*.

40.2. Hàm và đối số của PL/Tcl

Để tạo một hàm trong ngôn ngữ TL/Tcl, hãy sử dụng cú pháp tiêu chuẩn CREATE FUNCTION:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
        # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU là y hệt, ngoại trừ là ngôn ngữ này phải được chỉ định như là pltclu.

1 http://www.tcl.tk/

Thân hàm là một mẩu script Tcl đơn giản. Khi hàm được gọi, các giá trị đối số được truyền như các biến $1 ... $n tới script Tcl. Kết quả được trả về từ mã Tcl theo một cách thức thông thường, với một lệnh trả về.

Ví dụ, một hàm trả về nhiều hơn 2 giá trị số nguyên có thể được định nghĩa như sau:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
        if {$1 > $2} {return $1}
        return $2
$$ LANGUAGE pltcl STRICT;
```

Lưu ý cụm từ STRICT, nó giúp chúng ta khỏi phải nghĩ về các giá trị đầu vào null: nếu một giá trị null được truyền, thì hàm sẽ không được gọi hoàn toàn, nhưng sẽ chỉ trả về tự động kết quả null.

Trong một hàm không khắt khe, nếu giá trị thực của một đối số là null, thì biến tương ứng $n sẽ được thiết lập về một chuỗi rỗng. Để dò tìm xem liệu một đối số đặc biệt có là null hay không, hãy sử dụng hàm argisnull. Ví dụ, giả sử là chúng ta muốn tcl_max với một null và một không null để trả về đối số không null, thay vì null:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
        if {[argisnull 1]} {
                if {[argisnull 2]} { return_null }
                return $2
        }
        if {[argisnull 2]} { return $1 }
        if {$1 > $2} {return $1}
        return $2
$$ LANGUAGE pltcl;
```

Như được chỉ ra ở trên, để trả về một giá trị null từ một hàm PL/Tcl, hãy thực thi return_null. Điều này có thể được thực hiện bất chấp hàm là khắt khe hay không.

Các đối số dạng tổng hợp được truyền tới hàm như các mảng Tcl. Các tên phần tử của mảng là các tên thuộc tính của dạng tổng hợp đó. Nếu một thuộc tính trong hàng được truyền có giá trị null, thì nó sẽ không xuất hiện trong mảng đó. Đây là một ví dụ:

```
CREATE TABLE employee (
        name text,
        salary integer,
        age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
        if {200000.0 < $1(salary)} {
                return "t"
        }
        if {$1(age) < 30 && 100000.0 < $1(salary)} {
                return "t"
        }
        return "f"
$$ LANGUAGE pltcl;
```

Hiện không có sự hỗ trợ nào cho việc trả về một giá trị kết quả dạng tổng hợp, cũng không cho việc trả về các tập hợp.

PL/Tcl hiện không có hỗ trợ đầy đủ cho các dạng miền: nó đối xử với một miền y hệt như dạng vô

hướng nằm bên dưới. Điều này có nghĩa là các ràng buộc có liên quan tới miền đó sẽ không bị ép buộc. Đây không phải là một vấn đề đối với các đối số hàm, nhưng nó là một mối nguy nếu bạn khai báo một hàm PL/Tcl như việc trả về một dạng miền.

40.3. Giá trị dữ liệu trong PL/Tcl

Các giá trị đối số được cung cấp cho mã hàm PL/Tcl đơn giản là các đối số đầu vào được chuyển đổi sang dạng văn bản (hệt như nếu chúng từng được lệnh SELECT hiển thị). Ngược lại, lệnh return sẽ chấp nhận bất kỳ chuỗi nào là định dạng đầu vào truy cập được đối với dạng trả về được khai báo của hàm đó. Vì thế, trong hàm PL/Tcl, tất cả các giá trị chỉ là các chuỗi văn bản.

40.4. Dữ liệu tổng thể trong PL/Tcl

Đôi khi là hữu dụng để có vài dữ liệu tổng thể được nắm giữ giữa 2 lời gọi tới một hàm hoặc được chia sẻ giữa các hàm khác nhau. Điều này dễ dàng được thực hiện trong PL/Tcl, nhưng có vài hạn chế phải được hiểu.

Vì các lý do an toàn, PL/Tcl thực thi hàm được bất kỳ vai trò nào của SQL gọi trong một trình biên dịch Tcl riêng rẽ đối với vai trò đó. Điều này ngăn chặn được sự can thiệp ngẫu nhiên hoặc độc hại của một người sử dụng với hành vi của các hàm PL/Tcl của những người sử dụng khác. Từng trình biên dịch như vậy sẽ có các giá trị của riêng nó cho bất kỳ biến Tcl "tổng thể" nào. Vì thế, 2 hàm PL/Tcl sẽ chia sẻ cùng các biến tổng thể nếu và chỉ nếu chúng được cùng vai trò role SQL thực thi (thông qua các hàm SECURITY DEFINER functions, sử dụng SET ROLE, ...) bạn có thể cần thực hiện các bước rõ ràng để đảm bảo rằng các hàm PL/Tcl có thể chia sẻ dữ liệu. Để làm điều đó, hãy chắc chắn rằng các hàm mà sẽ giao tiếp sẽ được người sử dụng y hệt sở hữu, và đánh dấu chúng SECURITY DEFINER. Bạn tất nhiên phải cẩn thận rằng các hàm như vậy không thể được sử dụng để làm bất kỳ điều gì không mong muốn.

Tất cả các hàm PL/TclU được sử dụng trong một phiên làm việc thực thi trong cùng trình biên dịch Tcl, nó tất nhiên là phân biệt được với (các) trình biên dịch được sử dụng cho các hàm PL/Tcl. Vì thế dữ liệu tổng thể được tự động chia sẻ giữa các hàm PL/TclU. Điều này được xem là một rủi ro về an toàn vì tất cả các hàm PL/TclU thực thi ở mức tin cậy y hệt nhau, ấy là của một siêu người sử dụng cơ sở dữ liệu.

Để giúp bảo vệ các hàm PL/Tcl khỏi việc can thiệp không mong muốn với nhau, một mảng tổng thể được làm cho sẵn sàng đối với từng hàm thông qua lệnh upvar. Tên tổng thể của biến này là tên nội bộ của hàm, và tên cục bộ là GD. Được khuyến cáo rằng GD được sử dụng cho dữ liệu riêng nhất quán của một hàm. Sử dụng các biến tổng thể Tcl chỉ cho các giá trị mà bạn có ý định đặc biệt để chia sẻ giữa nhiều hàm. (Lưu ý là các mảng GD chỉ là tổng thể bên trong một trình biên dịch đặc biệt, vì thế chúng không bỏ qua được các hạn chế về an toàn được nêu ở trên).

Một ví dụ về việc sử dụng GD xuất hiện trong ví dụ spi_execp bên dưới.

40.5. Truy cập cơ sở dữ liệu từ PL/Tcl

Các lệnh sau là sẵn sàng để truy cập cơ sở dữ liệu từ thân của một hàm PL/Tcl:

spi_exec ?-count n? ?-array name? command ?loop-body?

Hãy thực thi một lệnh SQL được đưa ra như một chuỗi. Một lỗi trong lệnh là nguyên nhân để một lỗi sẽ nảy sinh. Nếu không, giá trị trả về của spi_exec là số hàng được xử lý (được chọn, được chèn, được cập nhật hoặc bị xóa) bởi lệnh đó, hoặc zero nếu lệnh đó là một lệnh tiện ích. Hơn nữa, nếu lệnh đó là một lệnh SELECT, thì các giá trị của các cột được chọn được đặt trong các biến Tcl như được mô tả bên dưới.

Giá trị tùy chọn -count nói cho spi_exec số lượng hàng tối đa phải xử lý trong lệnh đó. Hiệu quả của điều này là có thể so sánh được cho việc thiết lập truy vấn như một con trỏ và sau đó nói FETCH n.

Nếu lệnh đó là một lệnh SELECT, thì các giá trị của các cột kết quả được đặt trong tên các biến của Tcl sau các cột đó. Nếu lựa chọn *-array* được đưa ra, thì các giá trị cột đó thay vào đó được lưu trữ trong mảng được đặt tên có liên quan, với các tên cột được sử dụng như là các chỉ số mảng.

Nếu lệnh đó là một lệnh SELECT và không có script *loop-body* nào được đưa ra, thì chỉ hàng đầu tiên của các kết quả sẽ được lưu giữ trong các biến Tcl; các hàng còn lại, nếu có, sẽ bị bỏ qua. Không việc lưu trữ nào xảy ra nếu truy vấn đó không trả về hàng nào. (Trường hợp này có thể được việc kiểm tra kết quả của spi_exec dò tìm ra). Ví dụ:

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

sẽ thiết lập biến $cnt của Tcl về số các hàng trong catalog hệ thống pg_proc.

Nếu đối số tùy chọn *loop-body* được đưa ra, thì đó là một mẩu script Tcl mà được thực thi một lần cho từng hàng trong kết quả truy vấn đó. (*loop-body* bị bỏ qua nếu lệnh được đưa ra đó không phải là lệnh SELECT). Các giá trị các cột của hàng hiện hành được lưu trữ trong các biến của Tcl trước từng sự lặp. Ví dụ:

```
spi_exec -array C "SELECT * FROM pg_class" {
        elog DEBUG "have table $C(relname)"
}
```

sẽ in một thông điệp lưu ký cho từng hàng của pg_class. Tính năng này làm việc tương tự như các cấu trúc lặp Tcl khác; đặc biệt continue và break làm việc theo cách thông thường bên trong thân vòng lặp.

Nếu một cột kết quả truy vấn bằng null, thì biến đích cho nó là "unset" (không được thiết lập) hơn là được thiết lập.

spi_prepare *query typelist*

Chuẩn bị và lưu một kế hoạch truy vấn để thực thi sau này. Kế hoạch được lưu sẽ được giữ lại trong suốt toàn bộ phiên làm việc hiện hành.

Truy vấn đó có thể sử dụng các tham số, đó là, các nơi giữ chỗ cho các giá trị sẽ được cung cấp bất kỳ khi nào kế hoạch đó thực sự được thực thi. Trong chuỗi truy vấn đó, hãy tham chiếu tới các tham số của các dấu $1 ... $n. Nếu truy vấn đó sử dụng các tham số, thì tên các

dạng tham số đó phải được đưa ra như là một danh sách Tcl. (Hãy viết một danh sách rỗng cho *typelist* nếu không tham số nào được sử dụng).

Giá trị trả về từ spi_prepare là một ID truy vấn sẽ được sử dụng trong các lời gọi tiếp sau tới spi_execp. Xem spi_execp để có một ví dụ.

spi_execp ?-count n? ?-array name? ?-nulls string? queryid ?value-list? ?loop-body?

Hãy thực thi một truy vấn được chuẩn bị trước với spi_prepare. *queryid* là ID được spi_prepare trả về. Nếu truy vấn đó tham chiếu tới các tham số, thì value-list phải được cung cấp. Đây là một danh sách Tcl các giá trị thực cho các tham số đó. Danh sách đó phải có độ dài y hệt như danh sách dạng tham số được đưa ra trước đó cho spi_prepare. Hãy bỏ qua *value-list* nếu truy vấn không có các tham số.

Giá trị tùy chọn cho *-nulls* là một chuỗi không gian và các ký tự 'n' nói cho spi_execp tham số nào sẽ là các giá trị null. Nếu được đưa ra, nó phải có độ dài chính xác y hệt như *value-list*. Nếu nó không được đưa ra, thì tất cả các giá trị tham số là không null (nonnull).

Ngoại trừ cách thức ở đó truy vấn và các tham số của nó được chỉ định, spi_execp làm việc hệt như like spi_exec. Các lựa chọn -count, -array, và loop-body là y hệt nhau, và vì thế cả giá trị kết quả.

Đây là ví dụ về một hàm PL/Tcl đang sử dụng một kế hoạch được chuẩn bị:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
        if {![ info exists GD(plan) ]} {
                # prepare the saved plan on the first call
                set GD(plan) [ spi_prepare \
                        "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                        [ list int4 int4 ] ]
        }
        spi_execp -count 1 $GD(plan) [ list $1 $2 ]
        return $cnt
$$ LANGUAGE pltcl;
```

Chúng ta cần các dấu chéo ngược bên trong chuỗi truy vấn được đưa ra cho spi_prepare để đảm bảo rằng các dấu $n sẽ được truyền qua tới spi_prepare như cần thiết, và không bị sự thay thế biến Tcl thay thế.

spi_lastoid

Trả về OID hàng được spi_exec hoặc spi_execp mới nhất chèn vào, nếu lệnh đó từng là INSERT hàng duy nhất và bảng được sửa đổi có chứa các OID. (Nếu không, bạn sẽ có zero).

quote string

Đúp bản tất cả các trường hợp các ký tự dấu ngoặc đơn và dấu chéo ngược trong chuỗi được đưa ra. Điều này có thể được sử dụng để trích dẫn an toàn các chuỗi sẽ được chèn vào SQL được đưa ra cho spi_exec hoặc spi_prepare. Ví dụ, hãy nghĩ về chuỗi lệnh SQL như:

```
"SELECT '$val' AS ret"
```

trong đó biến Tcl val thực sự chứa doesn't. Điều này có thể gây ra chuỗi lệnh cuối cùng:

```
SELECT 'doesn't' AS ret
```

điều có thể gây là một lỗi phân tích cú pháp trong quá trình spi_exec or spi_prepare. Để làm

đúng, lệnh được đệ trình phải có:

```
SELECT 'doesn"t' AS ret
```

nó có thể được thực hiện trong PL/Tcl bằng việc sử dụng:

```
"SELECT '[ quote $val ]' AS ret"
```

Một ưu điểm của spi_execp là bạn không phải trích dẫn các giá trị tham số giống thế này, vì các tham số không bao giờ đi qua được như một phần của một chuỗi lệnh SQL.

elog level msg

Phát ra một thông điệp lưu ký hoặc lỗi. Các mức có khả năng là DEBUG, LOG, INFO, NOTICE, WARNING, ERROR, và FATAL. ERROR làm phát sinh một điều kiện lỗi; nếu điều này không được mã Tcl xung quanh bắt được, thì lỗi sẽ nhân ra tới truy vấn cuộc gọi, làm cho giao dịch hoặc giao dịch con hiện hành sẽ bị bỏ qua. Điều này có hiệu quả y hệt như lệnh error của Tcl. FATAL bỏ qua giao dịch đó và làm cho phiên làm việc hiện hành bị tắt. (Có thể là lý do không tốt để sử dụng mức lỗi này trong các hàm PL/Tcl, nhưng nó được cung cấp vì tính hoàn chỉnh). Các mức khác chỉ sinh ra các thông điệp các mức độ ưu tiên khác nhau. Liệu các thông điệp có ưu tiên đặc biệt có được nêu cho máy trạm, được viết tới lưu ký máy chủ, hoặc cả 2 được các biến cấu hình thông điệp log_min_messages hoặc client_min_messages kiểm soát hay không. Xem Chương 18 để có thêm thông tin.

40.6. Thủ tục trigger trong PL/Tcl

Các thủ tục trigger có thể được viết trong PL/Tcl. PostgreSQL đòi hỏi rằng một thủ tục sẽ được gọi như một trigger phải được khai báo như một hàm không với các đối số và một dạng trả về trigger.

Thông tin từ trình quản lý trigger được truyền tới thân thủ tục đó trong các biến sau:

$TG_name

Tên của trigger từ lệnh CREATE TRIGGER.

$TG_relid

ID đối tượng của bảng làm cho thủ tục trigger đó được triệu gọi.

$TG_table_name

Tên bảng làm cho thủ tục trigger đó được triệu gọi.

$TG_table_schema

Sơ đồ của bảng đã làm cho thủ tục trigger đó được triệu gọi.

$TG_relatts

Danh sách Tcl các tên cột của bảng, có tiền tố là một phần tử của danh sách rỗng. Vì thế việc tra một tên cột trong danh sách với lệnh lsearch của Tcl trả về số phần tử bắt đầu bằng 1 cho cột thứ nhất, cách y hệt các cột sẽ được đánh số tùy ý trong PostgreSQL. (Các phần tử danh sách rỗng cũng xuất hiện trong các vị trí các cột đã bị bỏ, sao cho việc đánh số thuộc tính là đúng đối với các cột ở bên phải của chúng)

$TG_when

Chuỗi BEFORE hoặc AFTER phụ thuộc vào dạng sự kiện trigger.

$TG_level

Chuỗi ROW hoặc STATEMENT phụ thuộc vào dạng sự kiện trigger.

$TG_op

Chuỗi INSERT, UPDATE, DELETE, hoặc TRUNCATE phụ thuộc vào dạng sự kiện trigger.

$NEW

Một mảng có liên quan chứa các giá trị hàng của bảng mới cho các hành động UPDATE hoặc DELETE actions, hoặc rỗng cho INSERT. Mảng được đánh chỉ số theo tên cột. Các cột là null sẽ không xuất hiện trong mảng đó. Điều này không được thiết lập cho các trigger mức lệnh.

$OLD

Một mảng có liên quan chứa các giá trị hàng của bảng cũ cho các hành động UPDATE hoặc DELETE, hoặc cho INSERT. Mảng đó được đánh chỉ số theo tên cột. Các cột mà là null sẽ không xuất hiện trong mảng đó. Đây không là tập hợp cho các trigger mức lệnh.

$args

Một danh sách các đối số của Tcl cho thủ tục được đưa ra trong lệnh CREATE TRIGGER. Các đối số đó cũng có khả năng truy cập được như $1 ... $n trong thân thủ tục.

Giá trị trả về từ một thủ tục trigger có thể là một trong các chuỗi OK hoặc SKIP, hoặc một danh sách như được lệnh array get của Tcl trả về. Nếu giá trị trả về là OK, thì hoạt động (INSERT / UPDATE / DELETE) mà làm bật dậy trigger sẽ tiếp tục bình thường. SKIP nói cho trình quản lý trigger âm thầm ép hoạt động cho hàng đó. Nếu một danh sách được trả về, thì nó nói cho PL/Tcl trả về một hàng được sửa đổi tới trình quản lý trigger mà sẽ được chèn vào thay vì của trình quản lý trigger được đưa ra trong $NEW. (Điều này làm việc chỉ cho INSERT và UPDATE). Không nhất thiết nói rằng tất cả điều này chỉ có nghĩa khi trigger là BEFORE và FOR EACH ROW; nếu không thì giá trị trả về bị bỏ qua.

Đây là một ví dụ thủ tục trigger mà ép một giá trị số nguyên trong một bảng bám theo số các bản cập nhật được thực thi trong hàng đó. Đối với các hàng mới được chèn vào, thì giá trị đó được khởi tạo về 0 và sau đó tăng dần lên theo mỗi hoạt động cập nhật.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
        switch $TG_op {
                INSERT {
                        set NEW($1) 0
                }
                UPDATE {
                        set NEW($1) $OLD($1)
                        incr NEW($1)
                }
                default {
                        return OK
                }
        }
        return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
        FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Lưu ý rằng bản thân thủ tục trigger không biết tên cột; điều đó được cung cấp từ các đối số trigger. Điều này cho phép thủ tục trigger được sử dụng lại với các bảng khác nhau.

40.7. Module và lệnh unknown

PL/Tcl có hỗ trợ cho việc tải tự động mã Tcl khi sử dụng. Nó nhận biết được một bảng đặc biệt, pltcl_modules, nó được giả thiết có chứa các module mã Tcl. Nếu bảng này tồn tại, module unknown được lấy từ bảng và được tải ngay lập tức vào trình biên dịch Tcl trước sự thực thi đầu tiên một hàm PL/Tcl trong một phiên làm việc của cơ sở dữ liệu. (Điều này xảy ra riêng rẽ đối với từng trình biên dịch Tcl, nếu hơn 1 trình biên dịch được sử dụng trong một phiên làm việc; xem Phần 40.4).

Trong khi module unknown có thể thực sự có bất kỳ script khởi tạo nào bạn cần, thì nó thường định nghĩa một thủ tục unknown của Tcl được triệu gọi bất kỳ khi nào Tcl không nhận biết được một tên thủ tục được triệu gọi. Phiên bản tiêu chuẩn của PL/Tcl đối với thủ tục này cố tìm một module trong pltcl_modules mà sẽ định nghĩa thủ tục được yêu cầu. Nếu một thủ tục được tìm ra, nó được tải vào trình biên dịch, và sau đó sự thực thi được cho phép để tiến hành với lời gọi thủ tục ban đầu. Một bảng thứ 2 pltcl_modfuncs đưa ra một chỉ số của các hàm nào được xác định bằng module nào, sao cho sự tra cứu là nhanh hợp lý.

Phân phối PostgreSQL bao gồm các script hỗ trợ để duy trì các bảng đó: pltcl_loadmod, pltcl_listmod, pltcl_delmod cũng như nguồn cho module tiêu chuẩn unknown trong share/unknown.pltcl. Module này phải được tải vào từng cơ sở dữ liệu để ban đầu hỗ trợ cho cơ chế tải tự động.

Các bảng pltcl_modules và pltcl_modfuncs phải đọc được đối với tất cả, nhưng là khôn ngoan để làm cho chỉ người quản trị cơ sở dữ liệu sở hữu và viết được chúng. Như một sự cảnh báo về an toàn, PL/Tcl sẽ bỏ qua pltcl_modules (và vì thế, không có ý định tải module unknown) trừ phi nó được một siêu người sử dụng sở hữu. Nhưng các quyền ưu tiên cập nhật trong bảng này có thể được trao cho những người sử dụng khác, nếu bạn tin tưởng họ đáng kể.

40.8. Tên thủ tục Tcl

Trong PostgreSQL, tên hàm y hệt có thể được sử dụng cho các định nghĩa hàm khác nhau miễn là số các đối số các dạng của chúng khác nhau. Tuy nhiên, Tcl đòi hỏi tất cả các tên thủ tục sẽ là phân biệt được.

PL/Tcl làm việc với điều này bằng việc làm cho các tên thủ tục Tcl nội bộ chứa ID đối tượng của tên y hệt và các dạng đối số khác sẽ cũng là các thủ tục Tcl khác. Đây là một mối lo không bình thường đối với một lập trình viên PL/Tcl, nhưng nó có thể là thấy được khi gỡ lỗi.

Chương 41. PL/Perl - Ngôn ngữ thủ tục Perl

PL/Perl là một ngôn ngữ thủ tục tải được mà cho phép bạn viết các hàm PostgreSQL trong ngôn ngữ[1] lập trình Perl.

Ưu điểm chính để sử dụng PL/Perl là vì điều này cho phép sử dụng, trong các hàm được lưu trữ, các toàn tử và các hàm "ép chuỗi" gấp bội có sẵn cho Perl. Việc phân tích cú pháp các chuỗi phức tạp có thể dễ dàng hơn bằng việc sử dụng Perl so với các hàm chuỗi và các cấu trúc kiểm soát được cung cấp trong PL/pgSQL.

Để cài đặt PL/Perl trong một cơ sở dữ liệu đặc biệt, hãy sử dụng createlang plperl dbname.

> **Mẹo**: nếu một ngôn ngữ được cài đặt vào template1, thì sau này tất cả các cơ sở dữ liệu được tạo ra sẽ có ngôn ngữ đó được cài đặt tự động.
>
> **Lưu ý**: Người sử dụng các gói nguồn phải đặc biệt cho phép xây dựng PL/Perl trong quá trình cài đặt. (Tham chiếu tới Chương 15 để có thêm thông tin). Người sử dụng các gói nhị phân có lẽ thấy PL/Perl trong một gói con riêng rẽ.

41.1. Hàm và đối số PL/Perl

Để tạo một hàm trong ngôn ngữ PL/Perl, hãy sử dụng cú pháp tiêu chuẩn CREATE FUNCTION:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
        # PL/Perl function body
$$ LANGUAGE plperl;
```

Thân hàm thường là mã Perl. Trên thực tế, mã gắn kết PL/Perl bao bọc nó trong một thủ tục con của Perl. Một hàm PL/Perl được gọi theo ngữ cảnh vô hướng, nên nó không thể trả về một danh sách. Bạn có thể trả về các giá trị không vô hướng (các mảng, các bản ghi, và các tập hợp) bằng việc trả về một tham chiếu, như được thảo luận bên dưới.

PL/Perl cũng hỗ trợ các khối mã nặc danh được gọi với lệnh DO:

```
DO $$
        # PL/Perl code
$$ LANGUAGE plperl;
```

Một khối mã nặc danh không nhận các đối số, và bất kể giá trị nào nó có thể trả về cũng bị bỏ qua. Nếu không thì nó hành xử hệt như một hàm.

> **Lưu ý**: Sử dụng các thủ tục con lồng nhau được đặt tên là nguy hiểm trong Perl, đặc biệt nếu chúng tham chiếu tới các biến từ vựng trong phạm vi bao bọc. Vì một hàm PL/Perl được gói trong một thủ tục con, bất kỳ thủ tục con nào được đặt tên mà bạn đặt bên trong cũng sẽ được lồng nhau. Nói chung, là rất nhanh để tạo các thủ tục con nặc danh mà bạn gọi qua một tham chiếu mã (coderef). Để có thêm thông tin, xem các khoản đầu vào cho Variable "%s" will not stay shared và Variable "%s" is not available trên trang chỉ dẫn perldiag, hoặc tìm kiếm trên Internet "perl nested named subroutine" (thủ tục con được đặt tên và lồng nhau của perl).

1 http://www.perl.org

Cú pháp của lệnh CREATE FUNCTION đòi hỏi thân hàm được viết như một hằng chuỗi. Thường là thuận tiện nhất để sử dụng dấu $ (xem Phần 4.1.2.4) cho hằng chuỗi. Nếu bạn chọn sử dụng cú pháp chuỗi thoát E'', thì bạn phải đúp bản bất kỳ dấu nháy đơn (') và dấu chéo ngược (\ ) nào được sử dụng trong thân hàm (xem Phần 4.1.2.1).

Các đối số và các kết quả được điều khiển như trong bất kỳ thủ tục con Perl nào khác: các đối số được truyền trong @_, và một giá trị kết quả được trả về với return hoặc như biểu thức cuối cùng được đánh giá trong hàm đó.

Ví dụ, một hàm trả về nhiều hơn 2 giá trị có thể được xác định như:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
        if ($_[0] > $_[1]) { return $_[0]; }
        return $_[1];
$$ LANGUAGE plperl;
```

Nếu giá trị null của SQL được truyền tới một hàm, thì giá trị đối số sẽ xuất hiện như "không xác định" (undefined) trong Perl. Định nghĩa hàm ở trên sẽ không hành xử rất tiện với các đầu vào null (trên thực tế, nó sẽ hành động như thể chúng là các zero). Chúng ta có thể thêm STRICT vào định nghĩa hàm để làm cho PostgreSQL thực hiện thứ gì đó hợp lý hơn: nếu một giá trị null được truyền, thì hàm sẽ không được gọi hoàn toàn, mà sẽ chỉ trả về kết quả null một cách tự động. Như một sự lựa chọn, chúng ta có thể kiểm tra các đầu vào không xác định trong thân hàm. Ví dụ, giả sử là chúng ta muốn perl_max với một đối số null và một đối số nonnull (không null) để trả về đối số nonnull, thay vì một giá trị null:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
        my ($x, $y) = @_;
        if (not defined $x) {
                return undef if not defined $y;
                return $y;
        }
        return $x if not defined $y;
        return $x if $x > $y;
        return $y;
$$ LANGUAGE plperl;
```

Như được chỉ ra ở trên, để trả về một giá trị null SQL từ một hàm PL/Perl, hãy trả về một giá trị không xác định. Điều này có thể được làm, bất kể là hàm có là khắt khe hay không.

Bất kỳ điều gì trong một đối số hàm mà không phải là một tham chiếu là một chuỗi, nó là trong trình bày văn bản bên ngoài theo tiêu chuẩn PostgreSQL cho dạng dữ liệu phù hợp. Trong trường hợp số thường hoặc các dạng văn bản, Perl sẽ chỉ làm đúng điều đó và lập trình viên sẽ thường không phải lo về nó. Tuy nhiên, trong các trường hợp khác đối số đó sẽ cần phải được chuyển đổi thành một dạng hữu dụng hơn trong Perl. Ví dụ, hàm decode_bytea có thể được sử dụng để chuyển đổi một đối số dạng bytea thành nhị phân không thoát được.

Tương tự, các giá trị được truyền ngược tới PostgreSQL phải là trong định dạng trình bày văn bản bên ngoài. Ví dụ, hàm encode_bytea có thể được sử dụng để thoát dữ liệu nhị phân đối với một giá trị trả về của dạng bytea.

Perl có thể trả về các mảng PostgreSQL như các tham chiếu tới các mảng Perl. Đây là 1 ví dụ:

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
        return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Các đối số dạng tổng hợp được truyền tới hàm như các tham chiếu tới các hàm băm. Các khóa của hàm băm là các tên thuộc tính của dạng tổng hợp. Đây là một ví dụ:

```
CREATE TABLE employee (
        name text,
        basesalary integer,
        bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
        my ($emp) = @_;
        return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

Một hàm PL/Perl có thể trả về một kết quả dạng tổng hợp bằng việc sử dụng tiếp cận y hệt: trả về một tham chiếu tới một hàm băm mà có các thuộc tính được yêu cầu. Ví dụ:

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
        return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Bất kỳ cột nào trong dạng dữ liệu kết quả được khai báo mà không hiện diện trong hàm băm đó sẽ được trả về như là các giá trị null.

Các hàm PL/Perl cũng có thể trả về các tập hợp các dạng tổng hợp hoặc vô hướng. Thường thì bạn sẽ muốn trả về các hàng cùng một lúc, cả để tăng tốc độ thời gian khởi đầu và cả để giữ hàng cho toàn bộ tập kết quả trong bộ nhớ. Bạn có thể làm điều này với return_next như được minh họa bên dưới. Lưu ý rằng sau return_next cuối cùng, bạn phải đặt hoặc return hoặc (tốt hơn) return undef.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
        foreach (0..$_[0]) {
                return_next($_);
        }
        return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
        return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
        return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
        return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
        return undef;
$$ LANGUAGE plperl;
```

Đối với các tập kết quả nhỏ, bạn có thể trả về một tham chiếu tới một mảng mà chứa hoặc các vô hướng, các tham chiếu tới các mảng, hoặc các tham chiếu tới các hàm băm cho các dạng đơn giản, các dạng mảng, và các dạng tổng hợp, một cách tương ứng. Đây là vài ví dụ đơn giản của việc trả về toàn bộ tập kết quả như một tham chiếu mảng:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
        return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
        return [
                { f1 => 1, f2 => 'Hello', f3 => 'World' },
                { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
                { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
        ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

Nếu bạn muốn sử dụng phương pháp pragma strict với mã của bạn thì bạn có một ít lựa chọn. Đối với sử dụng tổng thể tạm thời thì bạn có thể SET plperl.use_strict về đúng (true). Điều này sẽ ảnh hưởng tới các biên dịch sau này của các hàm PL/Perl, nhưng không ảnh hưởng tới các hàm được biên dịch rồi trong phiên làm việc hiện hành.

Đối với sử dụng tổng thể vĩnh viễn thì bạn có thể thiết lập plperl.use_strict về đúng (true) trong tệp postgresql.conf. Đối với sử dụng vĩnh viễn trong các hàm đặc biệt thì bạn có thể đơn giản đặt:

```
use strict;
```

ở đỉnh của thân hàm.

Phương pháp pragma feature cũng sẵn sàng để sử dụng nếu Perl của bạn là phiên bản 5.10.0 hoặc cao hơn.

41.2. Giá trị dữ liệu trong PL/Perl

Các giá trị đối số được cung cấp cho mã hàm của PL/Perl đơn giản là các đối số đầu vào được chuyển đổi thành dạng văn bản (hệt như nếu chúng đã từng được lệnh SELECT hiển thị). Ngược lại, các lệnh return và return_next sẽ chấp nhận bất kỳ chuỗi nào là định dạng đầu vào chấp nhận được cho dạng trả về được khai báo của hàm đó.

41.3. Hàm được xây dựng sẵn

41.3.1. Truy cập cơ sở dữ liệu từ PL/Perl

Truy cập tới bản thân cơ sở dữ liệu từ hàm Perl của bạn có thể được thực hiện qua các hàm sau:

```
spi_exec_query(query [, max-rows])
```

spi_exec_query thực thi một lệnh SQL và trả về toàn bộ tập hợp hàng như một tham chiếu tới một mảng các tham chiếu băm. *Bạn chỉ nên sử dụng lệnh này khi bạn biết rằng tập kết quả sẽ khá nhỏ*. Đây là một ví dụ về một truy vấn (lệnh SELECT) với số hàng tối đa tùy chọn:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

Điều này trả về tới 5 hàng từ bảng my_table. Nếu my_table có một cột my_column, bạn có thể có được giá trị đó từ hàng $i của kết quả giống thế này:

```
$foo = $rv->{rows}[$i]->{my_column};
```

Tổng số các hàng được trả về từ một truy vấn SELECT có thể được truy cập giống như thế này:

```
$nrows = $rv->{processed}
```

Đây là một ví dụ sử dụng dạng lệnh khác:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

Sau đó bạn có thể truy cập tình trạng lệnh (như, SPI_OK_INSERT) giống như thế này:

```
$res = $rv->{status};
```

Để có số hàng bị ảnh hưởng, hãy làm:

```
$nrows = $rv->{processed};
```

Đây là một ví dụ hoàn chỉnh:

```
CREATE TABLE test (
        i int,
        v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
        my $rv = spi_exec_query('select i, v from test;');
        my $status = $rv->{status};
        my $nrows = $rv->{processed};
        foreach my $rn (0 .. $nrows - 1) {
                my $row = $rv->{rows}[$rn];
                $row->{i} += 200 if defined($row->{i});
                $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
                return_next($row);
        }
        return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

spi_query(command)
spi_fetchrow(cursor)

spi_cursor_close(cursor)

spi_query và spi_fetchrow làm việc cùng nhau như một cặp các tập hợp hàng có thể là lớn, hoặc vì các trường hợp nơi mà bạn muốn trả về các hàng như khi chúng tới. spi_fetchrow làm việc chỉ với spi_query. Ví dụ sau minh họa cách mà bạn sử dụng chúng cùng nhau:

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
        use Digest::MD5 qw(md5_hex);
        my $file = '/usr/share/dict/words';
        my $t = localtime;
        elog(NOTICE, "opening file $file at $t" );
        open my $fh, '<', $file # ooh, it's a file access!
                or elog(ERROR, "cannot open $file for reading: $!");
        my @words = <$fh>;
        close $fh;
        $t = localtime;
        elog(NOTICE, "closed file $file at $t");
        chomp(@words);
        my $row;
        my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
        while (defined ($row = spi_fetchrow($sth))) {
                return_next({
                        the_num => $row->{a},
                        the_text => md5_hex($words[rand @words])
                });
        }
        return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

Thông thường, spi_fetchrow sẽ được lặp lại cho tới khi nó trả về undef, chỉ ra rằng không có nhiều hàng hơn để đọc. Con trỏ được spi_query trả về được tự động giải phóng khi spi_fetchrow trả về undef. Nếu bạn không muốn đọc tất cả các hàng, thì thay vào đó hãy gọi spi_cursor_close để giải phóng con trỏ. Làm như vậy mà hỏng thì sẽ có rò rỉ bộ nhớ.

spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)

spi_prepare, spi_query_prepared, spi_exec_prepared, và spi_freeplan triển khai chức năng y hệt nhưng là cho các truy vấn được chuẩn bị rồi. spi_prepare chấp nhận một chuỗi truy vấn với các chỗ giữ sẵn cho đối số được đánh số ($1, $2,...) và một chuỗi liệt kê các dạng đối số:

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

Một khi một kế hoạch truy vấn được một lời gọi tới spi_prepare chuẩn bị, thì kế hoạch đó có thể được sử dụng thay vì truy vấn chuỗi đó, hoặc trong spi_exec_prepared, nơi mà kết quả là y hệt như được trả về bởi spi_exec_query, hoặc trong in spi_query_prepared mà trả về một con trỏ chính xác như spi_query làm, nó có thể sau này được truyền tới spi_fetchrow. Tham số thứ 2 tùy chọn cho spi_exec_prepared là một tham chiếu các thuộc tính băm; thuộc tính duy nhất hiện được hỗ trợ là limit, nó thiết lập số lượng hàng tối đa được truy vấn trả về.

Ưu điểm của các truy vấn được chuẩn bị là có khả năng sử dụng một kế hoạch được chuẩn bị cho nhiều hơn một sự thực thi truy vấn. Sau khi kế hoạch không còn cần thiết nữa, nó có thể được giải phóng bằng spi_freeplan:

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                                'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

add_time    | add_time    | add_time
------------------+-----------------+------------------
2005-12-10 | 2005-12-11 | 2005-12-12
```

Lưu ý rằng chỉ số dưới của tham số đó trong spi_prepare được định nghĩa qua $1, $2, $3,..., vì thế tránh khai báo các chuỗi truy vấn trong các dấu ngoặc kép mà có thể dễ dẫn tới các lỗi khó bắt.

Một ví dụ khác minh họa sử dụng một tham số tùy biến trong spi_exec_prepared:

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                        FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                        WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();
query_hosts
```

```
-----------------
(1,192.168.1.1)
(2,192.168.1.2)
(2 rows)
```

41.3.2. Hàm tiện ích trong PL/Perl

elog(level, msg)

Phát một thông điệp lưu ký hoặc lỗi. Các mức có khả năng là DEBUG, LOG, INFO, NOTICE, WARNING, và ERROR. ERROR làm dấy lên một điều kiện lỗi; nếu điều này không được mã Perl bao quanh bẫy được, thì lỗi đó sẽ nhân giống ra tới truy vấn gọi, làm cho giao dịch hoặc giao dịch con sẽ bị bỏ. Điều này có hiệu lực y hệt như lệnh die của Perl. Các mức khác chỉ sinh ra các thông điệp các mức ưu tiên khác. Liệu các thông điệp của một ưu tiên đặc biệt có được nêu cho máy trạm hay không, được viết tới lưu ký máy chủ, hoặc cả 2 sẽ được các biến cấu hình log_min_messages và client_min_messages kiểm soát. Xem Chương 18 để có thêm thông tin.

quote_literal(string)

Trả về chuỗi được đưa ra được trích dẫn phù hợp để được sử dụng như một hằng chuỗi trong một chuỗi lệnh SQL. Các dấu nháy đơn và các dấu chéo ngược được nhúng sẽ được đúp bản đúng phù hợp. Lưu ý rằng quote_literal trả về không xác định (undef) ở đầu vào không xác định; nếu đối số có thể là không xác định, thì quote_nullable thường phù hợp hơn.

quote_nullable(string)

Trả về chuỗi được đưa ra được trích dẫn phù hợp sẽ được sử dụng như một hằng chuỗi trong một chuỗi lệnh SQL; hoặc, nếu đối số là không xác định, thì trả về chuỗi không trích dẫn "NULL". Các dấu nháy đơn và các dấu chéo ngược được nhúng sẽ được đúp bản phù hợp.

quote_ident(string)

Trả về chuỗi được đưa ra được trích dẫn phù hợp sẽ được sử dụng như một mã định danh trong một chuỗi lệnh SQL. Các dấu nháy sẽ chỉ được thêm vào nếu cần (như, nếu chuỗi chứa các ký tự không phải là mã định danh hoặc có thể là trường hợp được gấp). Các dấu nháy được nhúng sẽ được đúp bản đúng phù hợp.

decode_bytea(string)

Trả về dữ liệu nhị phân không thoát được do các nội dung của chuỗi được đưa ra đại diện, sẽ là bytea được mã hóa.

encode_bytea(string)

Trả về dạng bytea được mã hóa của các nội dung dữ liệu nhị phân của chuỗi được đưa ra.

encode_array_literal(array)
encode_array_literal(array, delimiter)

Trả về các nội dung của mảng được tham chiếu như một chuỗi ở định dạng hằng mảng (xem Phần 8.14.2).

Trả về giá trị đối số không tùy biến được nếu nó không là một tham chiếu tới một mảng. Dấu phân cách được sử dụng giữa các phần tử hằng mảng mặc định về ", " nếu một dấu phân cách không được chỉ định hoặc là không xác định.

encode_array_constructor(array)

Trả về các nội dung của mảng được tham chiếu như một chuỗi ở định dạng cấu trúc mảng (xem Phần 4.2.11). Các giá trị riêng rẽ được trích dẫn bằng việc sử dụng quote_nullable. Trả về giá trị đối số, được trích dẫn bằng việc sử dụng quote_nullable, nếu nó không phải là một tham chiếu tới một mảng.

looks_like_number(string)

Trả về một giá trị đúng (true) nếu nội dung của chuỗi được đưa ra trông giống một số, theo Perl, trả về sai (false) nếu khác. Trả về không xác định nếu đối số là không xác định. Việc dẫn và theo vết không gian bị bỏ qua. Inf và Infinity được coi như là các số.

41.4. Giá trị tổng thể trong PL/Perl

Bạn có thể sử dụng hàm băm tổng thể %_SHARED để lưu trữ dữ liệu, bao gồm cả các tham chiếu mã, giữa các lời gọi hàm cho vòng đời của phiên làm việc hiện hành.

Đây là một ví dụ đơn giản về các dữ liệu được chia sẻ:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
        if ($_SHARED{$_[0]} = $_[1]) {
                return 'ok';
        } else {
                return "cannot set shared variable $_[0] to $_[1]";
        }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
        return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl! How"s tricks?');
SELECT get_var('sample');
```

Đây là một ví dụ hơi phức tạp hơn có sử dụng một tham chiếu mã:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
        $_SHARED{myquote} = sub {
                my $arg = shift;
                $arg =~ s/(['\\])/\\$1/g;
                return "'$arg'";
        };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initializes the function */

/* Set up a function that uses the quote function */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
        my $text_to_quote = shift;
        my $qfunc = $_SHARED{myquote};
        return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(Bạn có thể đã thay thế thứ ở trên bằng sự trả về 1 dòng $_SHARED{myquote}->($_[0]); ở các chi phí của khả năng đọc).

Vì các lý do an toàn, PL/Perl thực thi các hàm được bất kỳ một vài trò nào của SQL gọi trong một trình biên dịch Perl riêng rẽ đối với vai trò đó. Điều này ngăn chặn được sự can thiệp ngẫu nhiên hoặc độc hại của một người sử dụng với hành vi của người sử dụng khác các hàm PL/Perl. Từng trình biên dịch như vậy có giá trị của riêng nó của biến %_SHARED và tình trạng tổng thể khác. Vì thế, 2 hàm PL/Perl sẽ chia sẻ cùng giá trị của %_SHARED nếu và chỉ nếu chúng được vai trò y hệt của SQL thực thi. Trong một ứng dụng nơi mà một phiên làm việc duy nhất thực thi mã dưới nhiều vai trò SQL (thông qua các hàm SECURITY DEFINER, sử dụng SET ROLE, ...) bạn có thể cần thực hiện các bước rõ ràng để đảm bảo rằng các hàm PL/Perl có thể chia sẻ dữ liệu qua %_SHARED. Để làm điều đó, phải chắc chắn rằng các hàm sẽ giao tiếp sẽ được người sử dụng y hệt sở hữu, và đánh dấu chúng SECURITY DEFINER. Bạn tất nhiên hãy thận trọng rằng các hàm như vậy không thể được sử dụng để làm bất kỳ điều gì không mong muốn.

41.5. PL/Perl tin cậy và không tin cậy

Thông thường, PL/Perl được cài đặt như một ngôn ngữ lập trình "tin cậy" có tên là plperl. Trong thiết lập này, các hoạt động nhất định của Perl bị vô hiệu hóa để giữ an toàn. Nói chung, các hoạt động bị giới hạn là các hoạt động tương tác với môi trường. Điều này bao gồm các hoạt động xử lý tệp, require và use (cho các module bên ngoài). Không có cách gì để truy cập các phần bên trong của tiến trình máy chủ cơ sở dữ liệu hoặc để giành được sự truy cập mức hệ điều hành bằng sự cho phép tiến trình của máy chủ, như một hàm C có thể làm. Vì thế, bất kỳ người sử dụng cơ sở dữ liệu không có quyền ưu tiên nào cũng có thể được cho phép để sử dụng ngôn ngữ này.

Đây là một ví dụ về một hàm sẽ không làm việc vì các hoạt động hệ thống tệp không được phép vì các lý do an toàn:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
        my $tmpfile = "/tmp/badfile";
        open my $fh, '>', $tmpfile
                or elog(ERROR, qq{could not open the file "$tmpfile": $!});
        print $fh "Testing writing to a file\n";
        close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
        return 1;
$$ LANGUAGE plperl;
```

Sự tạo ra hàm này sẽ hỏng khi nó sử dụng một hoạt động bị cấm và sẽ bị trình kiểm tra hợp lệ bắt.

Đôi khi là mong muốn để viết các hàm Perl mà sẽ không bị hạn chế. Ví dụ, người ta có thể muốn một hàm Perl gửi thư. Để xử trí các trường hợp đó, PL/Perl cũng có thể được cài đặt như một ngôn ngữ "không tin cậy" (thường được gọi là PL/PerlU). Trong trường hợp này ngôn ngữ Perl đầy đủ là sẵn sàng. Nếu chương trình createlang được sử dụng để cài đặt ngôn ngữ đó, thì tên ngôn ngữ đó plperlu sẽ chọn phương án PL/Perl không tin cậy đó.

Người viết một hàm PL/PerlU phải cẩn thận rằng hàm đó không thể được sử dụng để làm bất kỳ điều gì không mong muốn, vì nó sẽ có khả năng để làm bất kỳ điều gì có thể được một người sử dụng được đăng nhập vào như người quản trị cơ sở dữ liệu làm. Lưu ý rằng hệ thống cơ sở dữ liệu cho phép chỉ các siêu người sử dụng cơ sở dữ liệu mới được tạo các hàm trong các ngôn ngữ không

tin cậy.

Nếu hàm ở trên đã được một siêu người sử dụng tạo ra bằng việc sử dụng ngôn ngữ plperlu, thì sự thực thi có thể thành công.

Theo cách y hệt, các khối mã nặc danh được viết trong Perl có thể sử dụng các hoạt động có hạn chế nếu ngôn ngữ được chỉ định như là plperlu thay vì plperl, nhưng người gọi phải là một siêu người sử dụng.

> **Lưu ý**: Trong khi các hàm PL/Perl chạy trong một trình biên dịch Perl riêng rẽ cho từng vai trò SQL, thì tất cả các hàm PL/PerlU được thực thi trong một phiên làm việc được đưa ra chạy trong một trình biên dịch Perl duy nhất (nó không phải là bất kỳ ai được sử dụng đối với các hàm PL/Perl). Điều này cho phép các hàm PL/PerlU chia sẻ dữ liệu tự do, nhưng không giao tiếp nào có thể xảy ra giữa các hàm PL/Perl và PL/PerlU.

> **Lưu ý**: Perl không thể hỗ trợ nhiều trình biên dịch trong một tiến trình trừ phi nó đã được xây dựng với các cờ phù hợp, ấy là hoặc usemultiplicity hoặc useithreads. (usemultiplicity được ưu tiên hơn trừ phi bạn thực sự cần sử dụng các luồng. Để có thêm chi tiết, xem trang man perl nhúng). Nếu PL/Perl dược sử dụng với một bản sao của Perl mà đã không được xây dựng theo cách này, sau đó chỉ có khả năng có một trình biên dịch Perl cho từng phiên làm việc, và vì thế bất kỳ một phiên làm việc nào cũng chỉ có thể thực thi hoặc các hàm PL/PerlU, hoặc các hàm PL/Perl mà tất cả được gọi bằng vai trò SQL y hệt.

41.6. Trigger PL/Perl

PL/Perl có thể được sử dụng để viết các hàm trigger. Trong một hàm trigger, tham chiếu băm $_TD chứa thông tin về sự kiện trigger hiện hành. $_TD là một biến toàn thể, nó có một giá trị cục bộ riêng rẽ cho từng sự triệu gọi trigger. Các trường của tham chiếu băm $_TD là:

$_TD->{new}{foo}
: NEW giá trị của cột foo

$_TD->{old}{foo}
: OLD giá trị của cột foo

$_TD->{name}
: Tên của trigger đang được gọi

$_TD->{event}
: Sự kiện trigger: INSERT, UPDATE, DELETE, TRUNCATE, hoặc UNKNOWN

$_TD->{when}
: Khi trigger đã được gọi: BEFORE, AFTER, hoặc UNKNOWN

$_TD->{level}
: Mức chuỗi: ROW, STATEMENT, hoặc UNKNOWN

$_TD->{relid}

OID của bảng trong đó trigger được thực thi

`$_TD->{table_name}`

Tên của bảng trong đó trigger được thực thi

`$_TD->{relname}`

Tên của bảng trong đó trigger được thực thi. Điều này đã bị phản đối, và có thể bị loại bỏ trong một phiên bản trong tương lai. Xin hãy sử dụng `$_TD-> {table_name}` để thay.

`$_TD->{table_schema}`

Tên của sơ đồ theo đó bảng trong đó trigger được thực thi

`$_TD->{argc}`

Số các đối số của hàm trigger

`@{$_TD->{args}}`

Các đối số của hàm trigger. Không tồn tại nếu `$_TD->{argc}` bằng 0.

Các trigger mức hàng có thể trả về một trong những thứ sau đây:

`return;`

Thực thi hoạt động

`"SKIP"`

Không thực thi hoạt động

`"MODIFY"`

Chỉ ra rằng hàng `NEW` đã bị hàm trigger sửa đổi

Đây là ví dụ về một hàm trigger, minh họa vài điều được nêu ở trên:

```
CREATE TABLE test (
        i int,
        v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
        if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
                return "SKIP"; # skip INSERT/UPDATE command
        } elsif ($_TD->{new}{v} ne "immortal") {
                $_TD->{new}{v} .= "(modified by trigger)";
                return "MODIFY"; # modify row and execute INSERT/UPDATE command
        } else {
                return; # execute INSERT/UPDATE command
        }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
        BEFORE INSERT OR UPDATE ON test
        FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

41.7. Bên trên PL/Perl

41.7.1. Cấu hình

Phần này liệt kê các tham số cấu hình ảnh hưởng tới PL/Perl. Để thiết lập bất kỳ các tham số đó trước khi PL/Perl được tải lên, là cần thiết để đã thêm "`plperl`" vào danh sách

custom_variable_classes trong postgresql.conf.

plperl.on_init (string)

Chỉ định mã Perl sẽ được thực thi khi một trình biên dịch Perl lần đầu được khởi tạo, trước khi nó được chuyên môn hóa để plperl hoặc plperlu sử dụng. Các hàm SPI là không sẵn sàng khi mã này được thực thi. Nếu mã đó hỏng với một lỗi thì nó sẽ bỏ sự khởi tạo của trình biên dịch và nhân giống ra tới truy vấn gọi, làm cho giao dịch hoặc giao dịch con hiện hành sẽ bị bỏ.

Mã Perl bị hạn chế đối với một chuỗi đơn. Mã dài hơn có thể được đặt trong một module và được tải bằng chuỗi on_init. Các ví dụ:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Bất kỳ module nào được plperl.on_init tải lên, hoặc trực tiếp hoặc gián tiếp, sẽ là sẵn sàng để plperl sử dụng. Điều này có thể tạo ra một rủi ro về an toàn. Để xem các module nào đã được tải lên thì bạn có thể sử dụng:

```
DO 'elog(WARNING, join ", ", sort keys %INC)' language plperl;
```

Sự khởi tạo sẽ xảy ra trong bưu điện (postmaster) nếu thư viện plperl được đưa vào trong shared_preload_libraries, trong trường hợp đó sự cân nhắc dư thừa sẽ được đưa ra cho rủi ro của việc làm bất ổn định bưu điện. Lý do chính chỉ ở lúc bắt đầu bưu điện, và sẽ sẵn sàng ngay mà không có việc tải tổng chi phí trong các phiên làm việc của cơ sở dữ liệu riêng rẽ. Tuy nhiên, hãy ghi nhớ trong đầu rằng tổng chi phí tránh được chỉ vì trình biên dịch Perl đầu tiên được một phiên làm việc cơ sở dữ liệu sử dụng - hoặc PL/PerlU, hoặc PL/Perl đối với vai trò đầu tiên của SQL mà gọi một hàm PL/Perl. Bất kỳ trình biên dịch Perl bổ sung thêm nào được tạo ra trong một phiên làm việc cơ sở dữ liệu cũng sẽ phải thực thi plperl.on_init sau đó. Hơn nữa, trong Windows sẽ không có lưu bất kỳ thứ gì từ việc tải sẵn, vì trình biên dịch Perl được tạo trong tiến trình của bưu điện không nhân giống thành các tiến trình con.

Tham số này chỉ có thể được lập trong tệp postgresql.conf hoặc trong dòng lệnh máy chủ.

plperl.on_plperl_init (string)
plperl.on_plperlu_init (string)

Các tham số đó chỉ định mã Perl để được thực thi khi một trình biên dịch Perl được chuyên môn hóa cho plperl hoặc plperlu một cách tương ứng. Điều này sẽ xảy ra khi một hàm PL/Perl hoặc PL/PerlU lần đầu được thực thi trong một phiên làm việc cơ sở dữ liệu, hoặc khi một trình biên dịch bổ sung phải được tạo ra vì ngôn ngữ khác được gọi hoặc một hàm PL/Perl được gọi bằng một vai trò SQL mới. Điều này cho phép bất kỳ sự khởi tạo nào được plperl.on_init thực hiện. Các hàm SPI là không sẵn sàng khi mã này được thực thi. Mã Perl trong plperl.on_plperl_init được thực thi sau "việc khóa xuống" trình biên dịch, và vì thế nó chỉ có thể thực hiện các hoạt động tin cậy.

Nếu mã hỏng với một lỗi mà nó sẽ bỏ sự khởi tạo và nhân giống tới truy vấn gọi, làm cho

giao dịch hoặc giao dịch con hiện hành sẽ bị bỏ. Bất kỳ hành động nào được thực hiện rồi trong Perl cũng sẽ không được hoãn; tuy nhiên, trình biên dịch đó sẽ không được sử dụng lại lần nữa. Nếu ngôn ngữ đó được sử dụng một lần nữa thì sự khởi tạo sẽ được cố một lần nữa bên trong một trình biên dịch Perl tươi mới.

Chỉ những siêu người sử dụng mới có thể thay đổi được các thiết lập đó. Dù các thiết lập đó có thể được thay đổi bên trong một phiên làm việc, thì những thay đổi như vậy sẽ không ảnh hưởng tới các trình biên dịch Perl mà đã được sử dụng rồi để thực thi các hàm.

plperl.use_strict (boolean)

Khi thiết lập các biên dịch tiếp sau đúng đối với PL/Perl thì các hàm sẽ có phương pháp pragma strict được kích hoạt. Tham số này không ảnh hưởng tới các hàm được biên dịch rồi trong phiên làm việc hiện hành.

41.7.2. Hạn chế và thiếu tính năng

Các tính năng sau hiện đang thiếu đối với PL/Perl, nhưng chúng có thể có các đóng góp sẽ tới.

- Các hàm PL/Perl không thể gọi nhau trực tiếp.
- SPI còn chưa được triển khai đầy đủ.
- Nếu bạn đang lấy các tập hợp dữ liệu rất lớn bằng việc sử dụng spi_exec_query, thì bạn nên nhận thức được rằng chúng tất cả sẽ đi vào bộ nhớ. Bạn có thể tránh được điều này bằng việc sử dụng spi_query /spi_fetchrow như được minh họa trước đó.

  Một vấn đề tương tự xảy ra nếu một hàm trả về một tập hợp truyền một tập hợp lớn các hàng ngược về cho PostgreSQL qua return. Bạn cũng có thể tránh được vấn đề này bằng việc sử dụng return_next cho từng hàng được trả về, như được chỉ ra trước đó.
- Khi một phiên làm việc kết thúc bình thường, không vì một lỗi sống còn nào, thì bất kỳ khối END nào từng được định nghĩa sẽ được thực thi. Hiện hành không hành động nào khác được thực thi. Đặc biệt, các trình điều khiển tệp không được tự động xả và các đối tượng không bị tự động phá hủy.

Chương 42. PL/Python - Thủ tục Python

Ngôn ngữ thủ tục PL/Python cho phép các hàm PostgreSQL sẽ được viết trong ngôn ngữ Python[1].

Để cài đặt PL/Python trong một cơ sở dữ liệu đặc biệt, hãy sử dụng plpythonu dbname (mà cũng xem thêm Phần 42.1).

> **Mẹo**: Nếu một ngôn ngữ được cài đặt vào template1, thì tất cả các cơ sở dữ liệu được tạo ra sau đó sẽ có ngôn ngữ đó được cài đặt một cách tự động.

Như với PostgreSQL 7.4, PL/Python chỉ sẵn sàng như một ngôn ngữ "không tin cậy", nghĩa là nó không đưa ra bất kỳ cách thức hạn chế nào những gì người sử dụng có thể làm trong nó. Vì thế nó đã được đổi tên thành plpythonu. Phương án tin cậy của plpython có thể trở thành sẵn sàng trong tương lai, nếu một cơ chế thực thi an toàn mới được phát triển trong Python. Người viết một hàm trong PL/Python không tin cậy phải cẩn thận rằng hàm đó không thể được sử dụng để làm bất kỳ điều gì không mong muốn, vì nó sẽ có khả năng làm bất kỳ điều gì có thể được một người sử dụng đã đăng nhập vào như người quản trị cơ sở dữ liệu làm. Chỉ siêu người sử dụng có thể tạo ra các hàm trong các ngôn ngữ không tin cậy như plpythonu.

> **Lưu ý**: Người sử dụng các gói nguồn phải đặc biệt kích hoạt sự xây dựng PL/Python trong tiến trình cài đặt. (Hãy tham chiếu tới các chỉ dẫn cài đặt để có thêm thông tin). Người sử dụng các gói nhị phân có thể thấy PL/Python trong một gói con riêng rẽ.

42.1. Python 2 so với Python 3

PL/Python hỗ trợ cả các phương án ngôn ngữ Python 2 và Python 3. (Các chỉ dẫn cài đặt PostgreSQL có thể có nhiều thông tin chính xác hơn về các phiên bản phụ chính xác hỗ trợ Python). Vì các phương án ngôn ngữ Python 2 và Python 3 là không tương thích trong một vài khía cạnh quan trọng, sơ đồ biến đổi và việc đặt tên sau đây được PL/Python sử dụng để tránh trộn lẫn chúng:

- Ngôn ngữ PostgreSQL đã đặt tên plpython2u cho các triển khai PL/Python dựa vào phương án ngôn ngữ Python 2.
- Ngôn ngữ PostgreSQL đã đặt tên plpython3u cho các triển khai PL/Python dựa vào phương án ngôn ngữ Python 3.
- Ngôn ngữ đó đã đặt tên plpythonu cho các triển khai PL/Python dựa vào phương án ngôn ngữ Python mặc định, nó hiện là Python 2. (Mặc định này là độc lập với bất kỳ cài đặt Python cục bộ nào có thể coi là "mặc định" của chúng, ví dụ, những gì /usr/bin/python có thể là). Mặc định có thể sẽ được thay đổi đối với Python 3 trong một phiên bản PostgreSQL trong tương lai, phụ thuộc vào tiến trình chuyển đổi sang Python 3 trong cộng đồng Python.

Nó phụ thuộc vào cấu hình được xây dựng hoặc các gói được cài đặt bất kể PL/Python cho Python 2 hoặc Python 3 hoặc cả 2 đều có sẵn.

> **Mẹo**: Phương án được xây dựng phụ thuộc vào phiên bản Python nào đã được thấy trong

1 http://www.python.org

quá trình cài đặt hoặc phiên bản nào rõ ràng được thiết lập bằng việc sử dụng biến môi trường PYTHON; xem Phần 15.5. Để làm cho cả 2 phương án của PL/Python sẵn sàng trong một cài đặt, cây nguồn phải được thiết lập cấu hình và được xây dựng 2 lần.

Điều này dẫn tới sự sử dụng và chiến lược chuyển đổi sau:

- Người sử dụng đang tồn tại và người sử dụng hiện còn chưa quan tâm tới Python 3 sử dụng tên ngôn ngữ plpythonu và không phải thay đổi bất kỳ điều gì đối với tương lai có thể thấy trước được. Được khuyến cáo dần dần "chứng minh trong tương lai" mã thông qua sự chuyển đổi sang Python 2.6/2.7 để đơn giản hóa sự chuyển đổi cuối cùng sang Python 3.

  Trong thực tế, nhiều hàm PL/Python sẽ chuyển sang Python 3 với ít hoặc không có thay đổi.
- Người sử dụng mà biết họ có mã phụ thuộc Python 2 nặng nề và không có kế hoạch để bao giờ đó thay đổi nó có thể sử dụng tên ngôn ngữ plpython2u. Điều này sẽ tiếp tục làm việc trong tương lai rất xa, cho tới khi sự hỗ trợ Python 2 có thể hoàn toàn bị PostgreSQL bỏ.
- Người sử dụng mà muốn đi sâu vào Python 3 có thể sử dụng tên ngôn ngữ plpython3u, nó sẽ giữ làm việc vĩnh viễn theo các tiêu chuẩn hiện nay. Trong tương lai xa, khi Python 3 có thể trở thành mặc định, thì họ có thể muốn loại bỏ con số "3" vì những lý do thẩm mỹ.
- Daredevils, người muốn xây dựng một môi trường hệ điều hành chỉ Python 3, có thể thay đổi các scritp được xây dựng để làm cho plpythonu tương đương với plpython3u, giữ trong đầu rằng điều này có thể làm cho cài đặt của họ không tương thích với hầu hết phần còn lại của thế giới.

Xem thêm tài liệu Cái gì mới trong Python 3.0 (What's New In Python 3.0[2]) để có thêm thông tin về việc chuyển sang Python 3.

Là không được phép sử dụng PL/Python dựa vào Python 2 và PL/Python dựa vào Python 3 trong cùng một phiên làm việc, vì các biểu tượng trong các module động có thể hỏng, có thể dẫn tới hỏng tiến trình máy chủ PostgreSQL. Có một kiểm tra ngăn chặn việc trộn Python các phiên bản chính trong một phiên làm việc. Nó sẽ bỏ qua phiên làm việc nếu một sự không khớp được tìm thấy. Tuy nhiên, là có khả năng để sử dụng cả 2 phương án PL/Python trong cùng một cơ sở dữ liệu, từ các phiên làm việc tách biệt nhau.

42.2. Hàm PL/Python

Các hàm trong PL/Python được khai báo qua cú pháp tiêu chuẩn CREATE FUNCTION:

```
CREATE FUNCTION funcname (argument-list)
        RETURNS return-type
AS $$
        # PL/Python function body
$$ LANGUAGE plpythonu;
```

Thân hàm đơn giản là một script Python. Khi hàm được gọi, các đối số của nó được truyền như các phần tử của danh sách args; các đối số được đặt tên cũng được truyền như các biến thông thường tới

2 http://docs.python.org/py3k/whatsnew/3.0.html

script Python đó. Sử dụng các đối số được đặt tên thường là có khả năng đọc được hơn. Kết quả được trả về từ mã Python theo cách thường dùng, với return hoặc yield (trong trường hợp của một lệnh được kết quả thiết lập). Nếu bạn không đưa ra một giá trị trả về, thì Python trả về mặc định None. PL/Python dịch None của Python thành giá trị null SQL.

Ví dụ, một hàm trả về lớn hơn 2 số nguyên có thể được định nghĩa như là:

```
CREATE FUNCTION pymax (a integer, b integer)
        RETURNS integer
AS $$
        if a > b:
                return a
        return b
$$ LANGUAGE plpythonu;
```

Mã Python được đưa ra như là thân định nghĩa hàm được biến đổi thành một hàm Python. Ví dụ, điều ở trên làm cho:

```
def __plpython_procedure_pymax_23456():
        if a > b:
                return a
        return b
```

giả thiết rằng 23456 là OID được PostgreSQL chỉ định cho hàm đó.

Các đối số được thiết lập như các biến toàn thể. Vì phạm vi các quy tắc của Python, điều này có hệ quả tế nhị rằng một biến đối số không thể được tái chỉ định bên trong hàm cho giá trị của một biểu thức có liên quan tới bản thân tên biến đó, trừ phi biến đó được tái khai báo như là toàn thể trong khối đó. Ví dụ, thứ sau đây sẽ không làm việc:

```
CREATE FUNCTION pystrip(x text)
        RETURNS text
AS $$
        x = x.strip() # error
        return x
$$ LANGUAGE plpythonu;
```

vì việc chỉ định tới x làm cho x là một biến cục bộ cho toàn bộ khối đó, và vì thế x nằm ở phía bên phải của sự chỉ định tham chiếu tới một biến x cục bộ còn chưa được chỉ định, không phải tham số hàm PL/Python.

Sử dụng lệnh global, điều này có thể được thực hiện để làm việc:

```
CREATE FUNCTION pystrip(x text)
        RETURNS text
AS $$
        global x
        x = x.strip() # ok now
        return x
$$ LANGUAGE plpythonu;
```

Nhưng được khuyến cáo không dựa vào chi tiết triển khai này của PL/Python. Là tốt hơn để đối xử với các tham số hàm như là chỉ đọc.

42.3. Giá trị dữ liệu

Thường nói, mục tiêu của PL/Python là để cung cấp một ánh xạ "tự nhiên" giữa các thế giới PostgreSQL và Python. Điều này nêu các qui tắc ánh xạ dữ liệu được mô tả bên dưới.

42.3.1. Ánh xạ dạng dữ liệu

Các đối số hàm được chuyển đổi từ dạng PostgreSQL của chúng sang một dạng tương ứng của Python:

- PostgreSQL boolean được chuyển đổi thành Python bool.
- PostgreSQL smallint và int được chuyển đổi thành Python int. PostgreSQL bigint được chuyển đổi thành in trong Python 2 và thành int trong Python 3.
- PostgreSQL real, double, và numeric được chuyển đổi thành Python float. Lưu ý rằng đối với numeric thì điều này làm mất thông tin và có thể dẫn tới các kết quả không đúng. Điều này có thể được sửa trong một phiên bản trong tương lai.
- PostgreSQL bytea được chuyển đổi thành Python str trong Python 2 và thành bytes trong Python 3. Trong Python 2, chuỗi đó sẽ được đối xử như một tuần tự byte mà không có bất kỳ việc mã hóa ký tự nào.
- Tất cả các dạng dữ liệu khác, bao gồm các dạng chuỗi ký tự PostgreSQL, được chuyển đổi thành Python str. Trong Python 2, chuỗi này sẽ là trong mã hóa máy chủ PostgreSQL; trong Python 3, nó sẽ là một chuỗi Unicode như tất cả các chuỗi.
- Đối với các dạng dữ liệu không phải vô hướng, xem bên dưới.

Các giá trị trả về của hàm được chuyển đổi thành các dạng dữ liệu trả về được khai báo của PostgreSQL như sau:

- Khi dạng trả về của PostgreSQL là boolean, thì giá trị trả về đó sẽ được đánh giá về sự đúng đắn theo quy tắc của Python. Đó là, 0 và chuỗi rỗng là sai (false), nhưng lưu ý 'f' là đúng.
- Khi dạng trả về của PostgreSQL là bytea, thì giá trị trả về sẽ được chuyển đổi thành một chuỗi (Python 2) hoặc các byte (Python 3) bằng việc sử dụng thứ tương ứng của Python, builtins, với kết quả được chuyển đổi thành bytea.
- Đối với tất cả các dạng trả về khác của PostgreSQL, giá trị Python được trả về được chuyển thành một chuỗi bằng việc sử dụng builtin str của Python, và kết quả đó được truyền tới hàm đầu vào của dạng dữ liệu của PostgreSQL.

  Các chuỗi trong Python 2 được yêu cầu sẽ nằm trong việc mã hóa máy chủ PostgreSQL khi chúng được truyền tới PostgreSQL. Các chuỗi mà không hợp lệ trong việc mã hóa máy chủ hiện hành sẽ đưa ra một lỗi, nhưng không phải tất cả việc mã hóa không phù hợp có thể dò tìm ra được, nên các dữ liệu rác vẫn có thể còn khi điều này không được thực hiện đúng. Các chuỗi Unicode được chuyển thành mã hóa đúng một cách tự động, sao cho có thể là an toàn hơn và thuận tiện hơn để sử dụng chúng. Trong Python 3, tất cả các chuỗi là Unicode.
- Đối với các dạng dữ liệu không phải vô hướng, xem bên dưới.

Lưu ý rằng những sự không khớp về logic giữa dạng trả về được khai báo của PostgreSQL và dạng dữ liệu của Python đối với đối tượng thực sự trả về là không được đặt cờ; giá trị đó sẽ được chuyển đổi trong bất kỳ trường hợp nào.

> **Mẹo**: Các hàm PL/Python không thể trả về hoặc dạng RECORD hoặc SETOF RECORD. Một sự khắc phục là viết một hàm PL/pgSQL mà tạo ra một bảng tạm, nhờ nó gọi hàm PL/Python để điền bảng đó, và sau đó nhờ hàm PL/pgSQL trả về RECORD chung từ bảng tạm đó.

42.3.2. Null, None

Nếu một giá trị null SQL được truyền tới một hàm, thì giá trị đối số sẽ xuất hiện như là None trong Python. Ví dụ, định nghĩa hàm của pymax chỉ ra trong Phần 42.2 sẽ trả về câu trả lời sai đối với các đầu vào null. Chúng ta có thể thêm STRICT vào định nghĩa hàm để làm cho PostgreSQL làm mọi điều hợp lý hơn: nếu một giá trị null được truyền, thì hàm sẽ không được gọi hoàn toàn, nhưng sẽ chỉ trả về một kết quả null một cách tự động. Như một sự lựa chọn, chúng ta có thể kiểm tra đối với các đầu vào null trong thân hàm:

```
CREATE FUNCTION pymax (a integer, b integer)
        RETURNS integer
AS $$
        if (a is None) or (b is None):
                return None
        if a > b:
                return a
        return b
$$ LANGUAGE plpythonu;
```

Như được chỉ ra ở trên, để trả về một giá trị SQL null từ một hàm PL/Python, hãy trả về giá trị None. Điều này có thể được thực hiện bất chấp hàm là khắt khe hay không.

42.3.3. Mảng, danh sách

Các giá trị mảng SQL được truyền trong PL/Python như một danh sách Python. Để trả về một giá trị mảng ngoài một hàm PL/Python, hãy trả về 1 tuần tự Python, ví dụ 1 danh sách hoặc bộ dữ liệu:

```
CREATE FUNCTION return_arr()
        RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();

return_arr
----------------
{1,2,3,4,5}
(1 row)
```

Lưu ý rằng trong Python, các chuỗi là tuần tự, nó có thể có các hiệu ứng không mong muốn mà có thể là quen thuộc với các lập trình viên Python:

```
CREATE FUNCTION return_str_arr()
        RETURNS varchar[]
AS $$
return "hello"
```

```
$$ LANGUAGE plpythonu;

SELECT return_str_arr();

return_str_arr
--------------------
{h,e,l,l,o}
(1 row)
```

42.3.4. Dạng tổng hợp

Các đối số dạng tổng hợp được truyền tới hàm như việc ánh xạ của Python. Các tên phần tử của việc ánh xạ là các tên thuộc tính của dạng tổng hợp. Nếu một thuộc tính trong hàng được truyền có giá trị null, thì nó có giá trị `None` trong ánh xạ đó. Đây là một ví dụ:

```
CREATE TABLE employee (
        name text,
        salary integer,
        age integer
);

CREATE FUNCTION overpaid (e employee)
        RETURNS boolean
AS $$
        if e["salary"] > 200000:
                return True
        if (e["age"] < 30) and (e["salary"] > 100000):
                return True
        return False
$$ LANGUAGE plpythonu;
```

Có nhiều cách để trả về các dạng hàng hoặc tổng hợp từ một hàm Python. Các ví dụ sau giả thiết chúng ta có:

```
CREATE TYPE named_value AS (
        name text,
        value integer
);
```

Một kết quả tổng hợp có thể được trả về như:

Dạng tuần tự (một bộ dữ liệu hoặc danh sách, nhưng không là một tập hợp vì nó không có khả năng đánh chỉ số)

> Các đối tượng tuần tự được trả về phải có số các khoản y hệt như dạng kết quả tổng hợp có các trường. Khoản đó với chỉ số 0 được chỉ định cho trường đầu tiên của dạng tổng hợp, 1 cho trường thứ 2 và cứ như thế. Ví dụ:
>
> ```
> CREATE FUNCTION make_pair (name text, value integer)
>         RETURNS named_value
> AS $$
>         return [ name, value ]
>         # or alternatively, as tuple: return ( name, value )
> $$ LANGUAGE plpythonu;
> ```
>
> Để trả về một SQL null cho bất kỳ cột nào, hãy chèn `None` vào vị trí tương ứng.

Việc ánh xạ (từ điển)

Giá trị cho từng cột dạng kết quả được truy xuất từ ánh xạ với tên cột như là khóa. Ví dụ:

```
CREATE FUNCTION make_pair (name text, value integer)
        RETURNS named_value
AS $$
        return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

Bất kỳ cặp khóa/giá trị từ điển dư thêm nào cũng sẽ bị bỏ qua. Các khóa còn thiếu được đối xử như các lỗi. Để trả về một giá trị SQL null cho bất kỳ cột nào, hãy chèn `None` với tên cột tương ứng như khóa.

Đối tượng (bất kỳ đối tượng nào cung cấp phương pháp `__getattr__`)

Điều này làm việc y hệt như một việc ánh xạ. Ví dụ:

```
CREATE FUNCTION make_pair (name text, value integer)
        RETURNS named_value
AS $$
        class named_value:
           def __init__ (self, n, v):
                 self.name = n
                 self.value = v
        return named_value(name, value)

        # or simply
        class nv: pass
        nv.name = name
        nv.value = value
        return nv
$$ LANGUAGE plpythonu;
```

42.3.5. Hàm trả về tập hợp

Một hàm PL/Python cũng có thể trả về các tập hợp vô hướng hoặc các dạng tổng hợp. Có vài cách thức để đạt được điều này vì đối tượng được trả về được biến đổi nội bộ thành một bộ lặp. Các ví dụ sau giả thiết chúng ta có dạng tổng hợp:

```
CREATE TYPE greeting AS (
        how text,
        who text
);
```

Một kết quả tập hợp có thể được trả về từ:

Dạng tuần tự (bộ dữ liệu, danh sách, tập hợp)

```
CREATE FUNCTION greet (how text)
        RETURNS SETOF greeting
AS $$
        # return tuple containing lists as composite types
        # all other combinations work also
        return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

Bộ lặp - Iterator (bất kỳ đối tượng nào cung cấp các phương pháp `__iter__` và `next`)

```
CREATE FUNCTION greet (how text)
```

```
        RETURNS SETOF greeting
AS $$
        class producer:
          def __init__ (self, how, who):
                self.how = how
                self.who = who
                self.ndx = -1
          def __iter__ (self):
                return self

        def next (self):
                self.ndx += 1
                if self.ndx == len(self.who):
                        raise StopIteration
                return ( self.how, self.who[self.ndx] )
        return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Bộ sinh - Generator (yield)

```
CREATE FUNCTION greet (how text)
        RETURNS SETOF greeting
AS $$
        for who in [ "World", "PostgreSQL", "PL/Python" ]:
                yield ( how, who )
$$ LANGUAGE plpythonu;
```

Cảnh báo

Vì lỗi Python #1483133[3], vài phiên bản gỡ lỗi của Python 2.4 (được thiết lập cấu hình và được biên dịch với lựa chọn --with-pydebug) được biết sẽ làm hỏng máy chủ PostgreSQL khi sử dụng một bộ lặp để trả về một kết quả tập hợp. Các phiên bản chưa được vá của Fedora 4 chứa lỗi này. Nó không xảy ra trong các phiên bản sản xuất của Python hoặc trong các phiên bản được vá của Fedora 4.

42.4. Chia sẻ dữ liệu

Từ điển toàn thể SD là sẵn sàng để lưu trữ các dữ liệu giữa các lời gọi hàm. Biến này là dữ liệu tĩnh riêng tư. Từ điển toàn thể GD là dữ liệu công khai, sẵn có cho tất cả các hàm Python trong một phiên làm việc. Hãy sử dụng cẩn thận.

Mỗi hàm có môi trường thực thi của riêng nó trong trình biên dịch Python, sao cho các đối số hàm và dữ liệu toàn thể từ myfunc là không có sẵn cho myfunc2. Ngoại trừ là dữ liệu trong từ điển GD, như được nêu ở trên.

42.5. Khối mã nặc danh

PL/Python cũng hỗ trợ các khối mã nặc danh được gọi với lệnh DO:

```
DO $$
        # PL/Python code
```

```
$$ LANGUAGE plpythonu;
```

Một khối mã nặc danh không nhận các đối số, và bất kỳ giá trị nào nó có thể trả về cũng sẽ bị bỏ. Nếu không thì nó hành xử hệt như một hàm.

42.6. Hàm trigger

Khi một hàm được sử dụng như một trigger, từ điển TD chứa các giá trị có liên quan tới trigger:

TD["event"]
: chứa sự kiện như một chuỗi: INSERT, UPDATE, DELETE, TRUNCATE, hoặc UNKNOWN.

TD["when"]
: chứa một BEFORE, AFTER, hoặc UNKNOWN.

TD["level"]
: chứa một ROW, STATEMENT, hoặc UNKNOWN.

TD["new"]
TD["old"]
: Đối với một trigger mức hàng, thì 1 hoặc cả 2 trường đó chứa các hàng trigger tương ứng, phụ thuộc vào sự kiện của trigger đó.

TD["name"]
: chứa tên của trigger.

TD["table_name"]
: chứa tên bảng trong đó trigger xảy ra.

TD["table_schema"]
: chứa sơ đồ bảng trong đó trigger xảy ra.

TD["relid"]
: chứa OID của bảng trong đó trigger xảy ra.

TD["args"]
: Nếu lệnh CREATE TRIGGER bao gồm các đối số, thì chúng là sẵn sàng trong TD["args"][0] đối với TD["args"][n-1].

Nếu TD["when"] là BEFORE và TD["level"] là ROW, thì bạn có thể trả về None hoặc "OK" từ hàm Python để chỉ định hàng được sửa đổi, "SKIP" để bỏ sự kiện đó, hoặc "MODIFY" để chỉ bạn đã sửa đổi hàng đó. Nếu không thì giá trị trả về bị bỏ qua.

42.7. Truy cập cơ sở dữ liệu

Module ngôn ngữ PL/Python tự động nhập một module Python gọi là plpy. Các hàm và hằng trong module này là sẵn sàng cho bạn trong mã Python như là plpy.foo.

Module plpy cung cấp 2 hàm gọi là execute và prepare. Việc gọi plpy.execute với một chuỗi truy vấn và một đối số giới hạn tùy chọn làm cho truy vấn đó sẽ được chạy và kết quả sẽ được trả về trong

một đối tượng kết quả. Đối tượng kết quả mô phỏng một danh sách đối tượng từ điển. Đối tượng kết quả có thể được số hàng và tên cột truy cập. Nó có các phương pháp bổ sung: nrows trả về số hàng được truy vấn đó trả về, và status là giá trị trả về SPI_execute(). Đối tượng kết quả có thể được sửa đổi. Ví dụ:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

trả về tới 5 hàng từ my_table. Nếu my_table có cột my_column, thì nó có thể được truy cập như:

```
foo = rv[i]["my_column"]
```

Hàm thứ 2, plpy.prepare, chuẩn bị kế hoạch thực thi cho một truy vấn. Nó được gọi với một chuỗi truy vấn và một danh sách các dạng tham số, nếu bạn có các tham chiếu tham số trong truy vấn đó. Ví dụ:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", [ "text" ])
```

text là dạng biến bạn sẽ truyền qua cho $1. Sau việc chuẩn bị một lệnh, bạn sử dụng hàm plpy.execute để chạy nó:

```
rv = plpy.execute(plan, [ "name" ], 5)
```

Đối số thứ 3 là hạn chế và là tùy chọn.

Các tham số và truy vấn và các trường hàng kết quả được chuyển đổi giữa các dạng dữ liệu của PostgreSQL và Python như được mô tả trong Phần 42.3. Ngoại lệ là các dạng tổng hợp hiện không được hỗ trợ: Chúng sẽ bị từ chối khi các tham số truy vấn và được chuyển đổi thành các chuỗi khi xuất hiện trong một kết quả truy vấn. Như một sự khắc phục cho vấn đề đó sau này, truy vấn đó có thể đôi khi được viết lại sao cho kết quả dạng tổng hợp xuất hiện như một hàng kết quả thay vì như một trường của hàng kết quả. Như một sự lựa chọn, chuỗi kết quả có thể được phân tích cú pháp riêng rẽ bằng tay, nhưng tiếp cận này không được khuyến cáo vì nó không được chứng minh có tương lai.

Khi bạn chuẩn bị một kế hoạch sử dụng module PL/Python đó thì nó được tự động lưu giữ. Hãy đọc tài liệu SPI (Chương 43) về mô tả những gì điều này có nghĩa. Để sử dụng có hiệu quả điều này khắp các lời gọi hàm thì người ta cần sử dụng một trong các từ điển lưu trữ nhất quán SD hoặc GD (xem Phần 42.4). Ví dụ:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
        if SD.has_key("plan"):
                plan = SD["plan"]
        else:
                plan = plpy.prepare("SELECT 1")
                SD["plan"] = plan
        # rest of function
$$ LANGUAGE plpythonu;
```

42.8. Hàm tiện ích

Module plpy cũng cung cấp các hàm plpy.debug(msg), plpy.log(msg), plpy.info(msg), plpy.notice(msg), plpy.warning(msg), plpy.error(msg), và plpy.fatal(msg). plpy.error và plpy.fatal thực sự đưa ra một ngoại lệ

của Python mà, nếu không nắm bắt được, sẽ nhân giống ra tới truy vấn gọi, làm cho giao dịch hoặc giao dịch con hiện hành sẽ bị bỏ. raise plpy.Error(msg) và raise plpy.Fatal(msg) là tương đương với plpy.error và plpy.fatal, một cách tương ứng. Các hàm khác chỉ sinh ra các thông điệp được viết tới lưu ký máy chủ, hoặc cả 2 được các biến cấu hình log_min_messages và client_min_messages kiểm soát. Xem Chương 18 để có thêm thông tin.

42.9. Biến môi trường

Vài biến môi trường được trình biên dịch Python chấp nhận cũng có thể được sử dụng để gây ảnh hưởng tới hành vi của PL/Python. Chúng có thể cần phải được thiết lập trong môi trường của tiến trình máy chủ chính của PostgreSQL, ví dụ trong script khởi động. Các biến môi trường có sẵn phụ thuộc vào phiên bản Python; xem tài liệu Python để có các chi tiết. Tại thời điểm viết tài liệu này, các biến môi trường sau có tác động tới PL/Python; giả thiết một phiên bản Python phù hợp:

- PYTHONHOME
- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE

(Dường như là chi tiết triển khai Python vượt ra khỏi sự kiểm soát của PL/Python mà vài trong số các biến môi trường được liệt kê trên trang python man chỉ hiệu quả trong trình biên dịch dòng lệnh và không trong trình biên dịch Python được nhúng).

Chương 43. Giao diện lập trình máy chủ

Giao diện lập trình máy chủ - SPI (*Server Programming Interface*) trao cho những người viết các hàm C do người sử dụng định nghĩa khả năng chạy các lệnh SQL bên trong các hàm của họ. SPI là một tập hợp các hàm giao diện để đơn giản hóa truy cập tới trình phân tích cú pháp (parser), trình lên kế hoạch (planner) và trình thực thi (executor). SPI cũng thực hiện quản lý bộ nhớ.

> **Lưu ý**: Các ngôn ngữ thủ tục có sẵn đưa ra các biện pháp khác nhau để thực thi các lệnh SQL từ các thủ tục. Hầu hết các tiện ích đó dựa vào SPI, nên tài liệu này cũng có thể sử dụng cho những người sử dụng các ngôn ngữ đó.

Để tránh hiểu lầm chúng tôi sẽ sử dụng khái niệm "hàm" khi chúng tôi nói về các hàm giao diện SPI và "thủ tục" cho một hàm C do người sử dụng định nghĩa mà đang sử dụng SPI.

Lưu ý là nếu một lệnh được triệu gọi thông qua SPI mà hỏng, thì sự kiểm soát sẽ không được trả về cho thủ tục của bạn. Thay vào đó, giao dịch hoặc giao dịch con trong đó thủ tục của bạn thực thi sẽ bị quay về tình trạng cũ. (Điều này có thể dường như gây ngạc nhiên biết rằng các hàm SPI hầu hết có các qui ước trả về lỗi được ghi thành tài liệu. Tuy nhiên, các qui ước đó chỉ áp dụng cho các lỗi được dò tìm thấy bên trong bản thân các hàm SPI đó). Là có khả năng để phục hồi sự kiểm soát sau một lỗi bằng việc thiết lập giao dịch con của riêng bạn xung quanh các lời gọi SPI mà có thể hỏng. Hiện hành, điều này còn chưa được ghi thành tài liệu vì các cơ chế được yêu cầu vẫn còn thay đổi liên tục.

Các hàm SPI trả về một kết quả không âm nếu thành công (hoặc qua một giá trị số nguyên được trả về hoặc trong biến toàn thể `SPI_result`, như được mô tả bên dưới). Nếu có lỗi, một kết quả âm hoặc NULL sẽ được trả về. Các tệp mã nguồn mà sử dụng SPI phải bao gồm tệp đầu đề `executor/spi.h`.

43.1. Hàm giao diện

SPI_connect

Tên

`SPI_connect` - kết nối một thủ tục tới trình quản lý SPI

Tóm tắt

```
int SPI_connect(void)
```

Mô tả

`SPI_connect` mở một kết nối từ một triệu gọi thủ tục tới trình quản lý SPI. Bạn phải gọi hàm này nếu bạn muốn thực thi các lệnh qua SPI. Vài hàm SPI tiện ích có thể được gọi từ các thủ tục không được kết nối.

Nếu thủ tục của bạn được kết nối rồi, thì `SPI_connect` sẽ trả về mã lỗi `SPI_ERROR_CONNECT`. Điều

này có thể xảy ra nếu một thủ tục đã gọi SPI_connect trực tiếp các lời gọi thủ tục khác mà gọi SPI_connect. Trong khi các lời gọi đệ quy tới trình quản lý SPI được cho phép khi một lệnh SQL được gọi qua SPI triệu gọi hàm khác mà sử dụng SPI, thì các lời gọi lồng nhau trực tiếp tới SPI_connect và SPI_finish là bị cấm. (Xem SPI_push và SPI_pop).

Giá trị trả về

SPI_OK_CONNECT
: nếu thành công

SPI_ERROR_CONNECT
: nếu có lỗi

SPI_finish

Tên

SPI_finish - bỏ kết nối một thủ tục khỏi trình quản lý SPI

Tóm tắt

```
int SPI_finish(void)
```

Mô tả

SPI_finish đóng một kết nối đang tồn tại tới trình quản lý SPI. Bạn phải gọi hàm này sau khi hoàn tất các hoạt động SPI cần thiết trong quá trình triệu gọi hiện hành thủ tục của bạn. Tuy nhiên, bạn không cần lo về việc làm cho điều này xảy ra, nếu bạn bỏ giao dịch qua via elog(ERROR). Trong trường hợp đó SPI sẽ tự làm sạch mình một cách tự động.

Nếu SPI_finish được gọi mà không có kết nối hợp lệ, thì nó sẽ trả về SPI_ERROR_UNCONNECTED. Không có vấn đề cơ bản nào với điều này; nó có nghĩa là trình quản lý SPI không có gì để làm.

Giá trị trả về

SPI_OK_FINISH
: nếu bỏ kết nối đúng phù hợp

SPI_ERROR_UNCONNECTED
: nếu được gọi từ một thủ tục không kết nối được

SIP_push

Tên

SPI_push - đẩy kho SPI để cho phép sử dụng SPI đệ quy

Tóm tắt

void SPI_push(void)

Mô tả

SPI_push sẽ được gọi trước khi thực thi thủ tục khác mà bản thân nó có thể muốn sử dụng SPI. Sau SPI_push, SPI không còn trong một tình trạng "được kết nối" nữa, và các lời gọi hàm SPI sẽ bị từ chối, trừ phi một SPI_execute tươi mới được thực hiện. Điều này đảm bảo một sự tách bạch sạch sẽ giữa tình trạng SPI thủ tục của bạn và tình trạng của thủ tục khác mà bạn gọi. Sau khi thủ tục khác trả về, lời gọi SPI_pop để phục hồi truy cập về tình trạng SPI của riêng bạn.

Lưu ý rằng SPI_execute và các hàm có liên quan tự động làm điều tương đương SPI_push trước khi truyền sự kiểm soát ngược về máy thực thi SQL, nên sẽ là không cần thiết cho bạn phải lo về điều này khi sử dụng các hàm đó. Chỉ khi bạn đang trực tiếp gọi mã tùy ý mà có thể chứa các lời gọi SPI_connect mà bạn cần để đưa ra SPI_push và SPI_pop.

SPI_pop

Tên

SPI_pop - chọn kho SPI để trả về từ sử dụng SPI đệ quy

Tóm tắt

void SPI_pop(void)

Mô tả

SPI_pop chọn môi trường trước đó từ kho lời gọi SPI. Xem SPI_push.

SPI_execute

Tên

SPI_execute - thực thi một lệnh

Tóm tắt

int SPI_execute(const char * command, bool read_only, long count)

Mô tả

SPI_execute thực thi lệnh SQL được chỉ định để count các hàng. Nếu read_only là đúng, thì lệnh đó phải là chỉ đọc, và sự thực thi tổng chi phí là thứ gì đó được giảm xuống.

Hàm này chỉ có thể được gọi từ một thủ tục được kết nối.

Nếu count là zero thì lệnh đó được thực thi cho tất cả các hàng mà nó áp dụng tới. Nếu count là lớn hơn zero, thì nhiều hơn count hàng sẽ được truy xuất; sự thực thi dừng khi sự đếm đạt được, giống hệt như việc thêm một mệnh đề LIMIT vào truy vấn vậy. Ví dụ,

```
SPI_execute("SELECT * FROM foo", true, 5);
```

sẽ truy xuất nhiều nhất 5 hàng từ bảng. Lưu ý là một giới hạn như vậy chỉ có hiệu lực khi lệnh đó thực sự trả về các hàng. Ví dụ,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

chèn tất cả các hàng từ bar, bỏ qua tham số count. Tuy nhiên, với

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

nhiều nhất 5 hàng có thể được chèn vào, vì sự thực thi có thể dừng sau hàng kết quả thứ 5 RETURNING được truy xuất.

Bạn có thể truyền nhiều lệnh trong một chuỗi, nhưng sau này các lệnh không thể phụ thuộc vào sự tạo các đối tượng sớm hơn trong chuỗi đó, vì toàn bộ chuỗi sẽ được phân tích cú pháp và được lên kế hoạch trước khi sự thực thi bắt đầu. SPI_execute trả về kết quả cho lệnh được thực thi cuối cùng. Giới hạn count áp dụng cho từng lệnh một cách tách biệt (thậm chí dù chỉ kết quả cuối cùng sẽ thực sự được trả về). Giới hạn đó không được áp dụng cho bất kỳ lệnh ẩn nào được qui tắc đó sinh ra.

Khi read_only là sai (false), SPI_execute tăng dần đếm lệnh và tính toán một hình chụp mới trước khi thực thi từng lệnh trong chuỗi đó. Hình chụp thực sự không thay đổi nếu mức cô lập giao dịch hiện hành là SERIALIZABLE, nhưng trong chế độ READ COMMITTED thì bản cập nhật hình chụp cho phép từng lệnh thấy các kết quả của các giao dịch mới được đệ trình từ các phiên làm việc khác. Điều này là cơ bản cho hành vi nhất quán khi các lệnh đang sửa đổi cơ sở dữ liệu.

Khi read_only là đúng (true), SPI_execute sẽ không cập nhật hoặc hình chụp hoặc đếm lệnh, và nó chỉ cho phép các lệnh SELECT bình thường xuất hiện trong chuỗi lệnh. Các lệnh đó được thực thi bằng việc sử dụng hình chụp được thiết lập trước đó cho truy vấn xung quanh. Chế độ thực thi này là hơi

nhanh hơn so với chế độ đọc/ghi vì việc loại trừ tổng chi phí lệnh. Nó cũng cho phép các hàm thực sự *ổn định* sẽ được xây dựng: vì các thực thi kế tiếp tất cả sẽ sử dụng cùng y hệt hình chụp đó, sẽ không có sự thay đổi nào trong các kết quả.

Thường là không khôn ngoan để trộn các lệnh chỉ đọc và chỉ ghi vào trong một hàm duy nhất có sử dụng SPI; điều đó có thể gây ra hành vi rất lúng túng, vì các truy vấn chỉ đọc có thể không thấy các kết quả của bất kỳ bản cập nhật cơ sở dữ liệu nào được thực hiện bằng các truy vấn đọc - ghi.

Số các hàng thực sự cho lệnh (cuối cùng) nào đã được thực thi được trả về trong biến toàn thể SPI_processed. Nếu giá trị trả về của hàm là SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING hoặc SPI_OK_UPDATE_RETURNING, thì sau đó bạn có thể sử dụng con trỏ toàn thể SPITupleTable *SPI_tuptable để truy cập các hàng kết quả. Vài lệnh tiện ích (như EXPLAIN) cũng trả về các tập hợp hàng, và SPI_tuptable cũng sẽ chứa kết quả trong các trường hợp đó.

Cấu trúc được định nghĩa vì thế:

```
typedef struct
{
        MemoryContext tuptabcxt; /* memory context of result table */
        uint32 alloced; /* number of alloced vals */
        uint32 free; /* number of free vals */
        TupleDesc tupdesc; /* row descriptor */
        HeapTuple *vals; /* rows */
} SPITupleTable;
```

vals là một mảng các con trỏ tới các hàng. (Số các khoản đầu vào hợp lệ được SPI_processed đưa ra).

tupdesc là một trình mô tả hàng mà bạn có thể truyền tới các hàm SPI khi làm việc với các hàng, uptabcxt, alloced, và free sẽ là các trường bên trong không có ý định để các trình gọi SPI sử dụng.

SPI_finish giải phóng tất cả các SPITupleTable được phân bổ trong quá trình thủ tục hiện hành. Bạn có thể giải phóng một bảng kết quả đặc biệt trước đó, nếu bạn làm xong với nó, bằng việc gọi SPI_freetuptable.

Đối số

const char * command

chuỗi chứa lệnh để thực thi

bool read_only

đúng (true) đối với thực thi chỉ đọc

long count

số hàng cực đại để trả về, hoặc 0 cho không có giới hạn

Giá trị trả về

Nếu sự thực thi lệnh đã thành công thì một trong các giá trị sau (không âm) sẽ được trả về:

SPI_OK_SELECT

nếu một SELECT (chứ không phải SELECT INTO) đã được thực thi

SPI_OK_SELINTO

nếu một SELECT INTO đã được thực thi

SPI_OK_INSERT

nếu một INSERT đã được thực thi

SPI_OK_DELETE

nếu một DELETE đã được thực thi

SPI_OK_UPDATE

nếu một UPDATE đã được thực thi

SPI_OK_INSERT_RETURNING

nếu một INSERT RETURNING đã được thực thi

SPI_OK_DELETE_RETURNING

nếu một DELETE RETURNING đã được thực thi

SPI_OK_UPDATE_RETURNING

nếu một UPDATE RETURNING đã được thực thi

SPI_OK_UTILITY

nếu một lệnh tiện ích (như, CREATE TABLE) đã được thực thi

SPI_OK_REWRITTEN

nếu lệnh đã được viết lại trong dạng lệnh khác (như, UPDATE trở thành INSERT) theo quy tắc.

Nếu có lỗi, một trong những giá trị âm sẽ được trả về:

SPI_ERROR_ARGUMENT

nếu command là NULL hoặc count ít hơn 0

SPI_ERROR_COPY

nếu COPY TO stdout hoặc COPY FROM stdin đã được cố thử

SPI_ERROR_TRANSACTION

nếu một lệnh điều khiển giao dịch đã được cố thử (BEGIN, COMMIT, ROLLBACK, SAVEPOINT, PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED hoặc bất kỳ phương án nào từ đó).

SPI_ERROR_OPUNKNOWN

nếu dạng lệnh là không rõ (sẽ không xảy ra)

SPI_ERROR_UNCONNECTED

nếu được gọi từ một thủ tục không được kết nối

Lưu ý

Tất cả các hàm thực thi truy vấn thiết lập cả SPI_processed và SPI_tuptable (chỉ con trỏ, không phải các nội dung của cấu trúc). Hãy lưu 2 biến toàn thể đó trong các biến thủ tục cục bộ nếu bạn cần truy cập bảng kết quả của SPI_execute hoặc hàm thực thi truy vấn khác khắp các lời gọi sau này.

SPI_exec

Tên

SPI_exec - thực thi một lệnh đọc/ghi

Tóm tắt

int SPI_exec(const char * command, long count)

Mô tả

SPI_exec là y hệt như SPI_execute, với tham số sau read_only luôn được lấy như false.

Đối số

const char * command

chuỗi chứa lệnh để thực thi

long count

số hàng cực đại để trả về, hoặc 0 cho không giới hạn

Giá trị trả về

Xem SPI_execute.

SPI_execute_with_args

Tên

SPI_execute_with_args - thực thi một lệnh với các tham số nằm ngoài dòng

Tóm tắt

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Mô tả

SPI_execute_with_args thực thi một lệnh có thể bao gồm các tham chiếu tới các tham số được cung cấp bên ngoài. Văn bản lệnh tham chiếu tới một tham số như `$n`, và lời gọi phân công các dạng dữ liệu và giá trị cho từng ký hiệu như vậy. read_only và count có cùng y hệt sự diễn giải như trong SPI_execute.

Ưu điểm chính của thủ tục này so với SPI_execute là các giá trị dữ liệu có thể được chèn vào trong lệnh mà không có việc trích dẫn/thoát nặng nhọc, và vì thế với ít rủi ro hơn nhiều đối với các cuộc tấn công tiêm SQL.

Các kết quả tương tự có thể đạt được với SPI_prepare đi sau là SPI_execute_plan; tuy nhiên, khi sử dụng hàm này thì kế hoạch truy vấn được tùy biến về các giá trị tham số đặc biệt được cung cấp. Đối với sự thực thi truy vấn một lần, thì hàm này sẽ được ưu tiên. Nếu lệnh y hệt sẽ được thực thi với nhiều tham số khác nhau, thì phương pháp nào có thể là nhanh hơn, phụ thuộc vào chi phí lên kế hoạch lại so với lợi ích của các kế hoạch tùy biến.

Đối số

const char * command

chuỗi lệnh

int nargs

số các tham số đầu vào (`$1`, `$2`, ...)

Oid * argtypes

một mảng các giá trị tham số thực

const char * nulls

một mảng mô tả các tham số nào là null

nếu nulls là NULL thì SPI_execute_with_args giả thiết rằng không tham số nào là null.

bool read_only

true cho thực thi chỉ đọc

long count

số hàng cực đại trả về, hoặc 0 cho không giới hạn

Giá trị trả về

Giá trị trả về là y hệt như đối với SPI_execute.

SPI_processed và SPI_tuptable được thiết lập như trong SPI_execute nếu thành công.

SPI_prepare

Tên

SPI_prepare - chuẩn bị một kế hoạch cho một lệnh, còn chưa có việc thực thi nó

Tóm tắt

SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)

Mô tả

SPI_prepare tạo và trả về một kế hoạch thực thi cho lệnh được chỉ định, nhưng không thực thi lệnh đó. Hàm này sẽ chỉ được gọi từ một thủ tục được kết nối.

Khi lệnh y hệt hoặc tương tự sẽ được thực thi lặp đi lặp lại, thì có thể là ưu điểm để thực hiện việc lên kế hoạch chỉ một lần. SPI_prepare chuyển đổi một chuỗi lệnh thành một kế hoạch thực thi mà có thể được thực thi lặp đi lặp lại bằng việc sử dụng SPI_execute_plan.

Một lệnh được chuẩn bị có thể được tổng quát hóa bằng việc viết các tham số ($1, $2,...) tại chỗ của những gì có thể là các hằng số trong một lệnh thông thường. Các giá trị thực sự của các tham số sau đó được chỉ định khi SPI_execute_plan được gọi. Điều này cho phép lệnh được chuẩn bị sẽ được sử dụng đối với một dải rộng lớn hơn các tình huống so với có thể mà không có các tham số.

Kế hoạch đó được SPI_prepare trả về có thể chỉ được sử dụng trong triệu gọi thủ tục hiện hành, vì SPI_finish giải phóng bộ nhớ được phân bổ cho một kế hoạch. Nhưng một kế hoạch có thể được lưu giữ dài hơn bằng việc sử dụng hàm SPI_saveplan.

Đối số

const char * command

chuỗi lệnh

int nargs

số các tham số đầu vào ($1, $2,...)

Oid * argtypes

con trỏ tới một mảng có chứa các OID các dạng dữ liệu của các tham số đó

Giá trị trả về

SPI_prepare trả về một con trỏ không null cho một kế hoạch thực thi. Nếu có lỗi, NULL sẽ được trả về, và SPI_result sẽ được thiết lập về một trong các mã lỗi y hệt được SPI_execute sử dụng, ngoại trừ là nó được thiết lập về SPI_ERROR_ARGUMENT nếu lệnh là NULL, hoặc nếu nargs là nhỏ hơn 0, hoặc nếu nargs là lớn hơn 0 và argtypes là NULL.

Lưu ý

SPIPlanPtr được khai báo như một con trỏ tới một dạng cấu trúc mù mờ trong spi.h. Là không khôn ngoan để cố truy cập các nội dung của nó trực tiếp, vì điều đó làm cho mã của bạn nhiều khả năng hơn phải gián đoạn trong các phiên bản trong tương lai của PostgreSQL.

Có một nhược điểm cho việc sử dụng các tham số: vì trình lên kế hoạch không biết các giá trị sẽ được cung cấp cho các tham số đó, có thể tệ hơn cho các lựa chọn lên kế hoạch so với nó có thể làm cho một lệnh bình thường với tất cả các hằng nhìn thấy được.

SPI_prepare_cursor

Tên

SPI_prepare_cursor - chuẩn bị kế hoạch cho một lệnh, còn chưa thực thi nó

Tóm tắt

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

Mô tả

SPI_prepare_cursor là y hệt như SPI_prepare, ngoại trừ là nó cũng cho phép đặc tả của tham số "các lựa chọn con trỏ" của trình lên kế hoạch. Đây là một mặt nạ bit có các giá trị được chỉ ra trong nodes/parsenodes.h cho trường options của DeclareCursorStmt. SPI_prepare luôn lấy các lựa chọn con trỏ là zero.

Đối số

const char * command

chuỗi lệnh

int nargs

số các tham số đầu vào ($1, $2,...)

Oid * argtypes

con trỏ tới một mảng chứa các OID các dạng dữ liệu của các tham số

int cursorOptions

mặt nạ bit số nguyên của các lựa chọn con trỏ; zero tạo ra hành vi mặc định

Giá trị trả về

SPI_prepare_cursor có các qui ước trả về y hệt như SPI_prepare.

Lưu ý

Các bit hữu dụng để thiết lập trong cursorOptions bao gồm CURSOR_OPT_SCROLL, CURSOR_OPT_NO_SCROLL, và CURSOR_OPT_FAST_PLAN. Lưu ý đặc biệt: CURSOR_OPT_HOLD bị bỏ qua.

SPI_prepare_params

SPI_prepare_params - chuẩn bị một kế hoạch cho một lệnh, chưa thực thi lệnh

Tóm tắt

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Mô tả

SPI_prepare_params tạo và trả về một kế hoạch thực thi cho lệnh được chỉ định, nhưng chưa thực thi lệnh. Hàm này là tương đương với SPI_prepare_cursor, bổ sung thêm là trình gọi có thể chỉ định trình phân tích cú pháp móc vào các hàm để kiểm soát việc phân tích cú pháp các tham chiếu tham số bên ngoài.

Đối số

const char * command

chuỗi lệnh

ParserSetupHook parserSetup

Hàm thiết lập móc của trình phân tích cú pháp

void * parserSetupArg

truyền qua đối số cho parserSetup

int cursorOptions

mặt nạ bit số nguyên các lựa chọn của con trỏ; zero tạo ra hành vi mặc định

Giá trị trả về

SPI_prepare_params có cùng y hệt các qui ước trả về như SPI_prepare.

SPI_getargcount

Tên

SPI_getargcount - trả về số các đối số mà một kế hoạch cần được SPI_prepare chuẩn bị

Tóm tắt

int SPI_getargcount(SPIPlanPtr plan)

Mô tả

SPI_getargcount trả về số các đối số cần để thực thi một kế hoạch được SPI_prepare chuẩn bị.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (SPI_prepare trả về)

Giá trị trả về

Sự tính đếm các đối số được kỳ vọng cho plan. Nếu plan là NULL hoặc không hợp lệ, thì SPI_result được thiết lập về SPI_ERROR_ARGUMENT và -1 được trả về.

SPI_getargtypeid

Tên

SPI_getargtypeid - trả về OID dạng dữ liệu cho đối số của một kế hoạch được SPI_prepare chuẩn bị

Tóm tắt

Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)

Mô tả

SPI_getargtypeid trả về OID đại diện cho id dạng đối với đối số thứ argIndex của một kế hoạch được SPI_prepare chuẩn bị. Đối số đầu tiên là trong chỉ số zero.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

int argIndex

chỉ số của đối số dựa vào zero

Giá trị trả về

Mã id dạng đối số trong chỉ số được đưa ra. Nếu kế hoạch là NULL hoặc không hợp lệ, hoặc argIndex là nhỏ hơn 0 hoặc không nhỏ hơn số các đối số được khai báo cho plan, thì SPI_result được thiết lập về SPI_ERROR_ARGUMENT và InvalidOid được trả về.

SPI_is_cursor_plan

Tên

SPI_is_cursor_plan - trả về đúng (true) nếu một kế hoạch được SPI_prepare chuẩn bị có thể được sử dụng với SPI_cursor_open

Tóm tắt

bool SPI_is_cursor_plan(SPIPlanPtr plan)

Mô tả

SPI_is_cursor_plan trả về đúng nếu một kế hoạch được SPI_prepare chuẩn bị có thể được truyền như một đối số tới SPI_cursor_open, hoặc sai nếu điều đó không phải thế. Các chỉ tiêu là thứ mà plan thể hiện một lệnh duy nhất và lệnh này trả về các bộ dữ liệu cho trình gọi; ví dụ, SELECT được cho phép trừ phi nó chứa mệnh đề INTO, và UPDATE được cho phép chỉ nếu nó có chứa mệnh đề RETURNING.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

Giá trị trả về

đúng hoặc sai để chỉ định nếu kế hoạch đó có thể sản sinh ra một con trỏ hay không, với SPI_result được đặt về zero. Nếu không có khả năng để xác định câu trả lời (ví dụ, nếu plan là NULL hoặc không hợp lệ, hoặc nếu được gọi khi không được kết nối tới SPI), thì SPI_result được thiết lập cho mã lỗi phù hợp và false được trả về.

SPI_execute_plan

Tên

SPI_execute_plan - thực thi một kế hoạch được SPI_prepare chuẩn bị

Tóm tắt

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

Mô tả

SPI_execute_plan thực thi kế hoạch được SPI_prepare chuẩn bị. read_only và count có cùng sự diễn giải như trong SPI_execute.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

Datum * values

Mảng các giá trị tham số thực. Phải có độ dài y hệt như số các đối số của kế hoạch.

const char * nulls

Mảng mô tả các tham số nào là null. Phải có độ dài y hệt như số các đối số của kế hoạch. n chỉ một giá trị null (khoản đầu vào trong values sẽ bị bỏ qua); một khoảng trống chỉ một giá trị không null (nonnull) (khoản đầu vào trong values là hợp lệ).

Nếu nulls là NULL thì SPI_execute_plan giả thiết là không tham số nào là null.

bool read_only

true cho thực thi chỉ đọc

long count

số hàng cực đại để trả về, hoặc 0 nếu không có giới hạn

Giá trị trả về

Giá trị trả về là y hệt như đối với SPI_execute, với các kết quả lỗi (âm) có thể thêm vào sau đây:

SPI_ERROR_ARGUMENT

nếu plan là NULL hoặc không hợp lệ, hoặc count là ít hơn 0

SPI_ERROR_PARAM

nếu values là NULL và plan đã được chuẩn bị với vài tham số

SPI_processed và SPI_tuptable sẽ được thiết lập trong SPI_execute nếu thành công.

SPI_execute_plan_with_paramlist

Tên

SPI_execute_plan_with_paramlist - thực thi một kế hoạch được SPI_prepare chuẩn bị

Tóm tắt

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Mô tả

SPI_execute_plan_with_paramlist thực thi kế hoạch được SPI_prepare chuẩn bị. Hàm này là tương đương với SPI_execute_plan ngoại trừ là thông tin về các giá trị tham số sẽ được truyền tới truy vấn được trình diễn một cách khác. Đại diện ParamListInfo có thể là thuận tiện cho việc truyền các giá trị đã có sẵn rồi ở định dạng đó. Nó cũng hỗ trợ sử dụng các tập hợp tham số động qua các hàm móc được chỉ định trong ParamListInfo.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

ParamListInfo params

cấu trúc dữ liệu chứa các dạng và các giá trị tham số; NULL nếu không có

bool read_only

true đối với thực thi chỉ đọc

long count

số hàng cực đại trả về, hoặc 0 cho không giới hạn

Giá trị trả về

Giá trị trả về là y hệt như đối với SPI_execute_plan.

SPI_processed và SPI_tuptable được thiết lập trong SPI_execute_plan nếu thành công.

SPI_execp

Tên

SPI_execp - thực thi kế hoạch ở chế độ đọc/ghi

Tóm tắt

int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)

Mô tả

SPI_execp là y hệt như SPI_execute_plan, với tham số read_only của cái sau luôn được lấy là false.

Đối số

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

Datum * values

Mảng các giá trị tham số thực. Phải có độ dài y hệt như số các đối số của kế hoạch.

const char * nulls

Mảng mô tả các tham số nào là null. Phải có độ dài y hệt như số các đối số của kế hoạch. n chỉ một giá trị null (khoản đầu vào values sẽ bị bỏ qua); một không gian chỉ một giá trị không null (nonnull) (khoản đầu vào trong values là hợp lệ).

Nếu nulls là NULL thì SPI_execp giả thiết rằng không tham số nào là null.

long count

số lượng cực đại các hàng trả về, hoặc 0 nếu không có giới hạn

Giá trị trả về

Xem SPI_execute_plan.

SPI_processed và SPI_tuptable được thiết lập như trong SPI_execute nếu thành công.

SPI_cursor_open

Tên

SPI_cursor_open - thiết lập một con trỏ sử dụng một kế hoạch được tạo ra với SPI_prepare

Tóm tắt

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

Mô tả

SPI_cursor_open thiết lập một con trỏ (nội bộ, một cổng) mà sẽ thực thi kế hoạch được SPI_prepare chuẩn bị. Các tham số có cùng ý nghĩa như các tham số tương ứng với SPI_execute_plan.

Sử dụng con trỏ thay cho việc thực thi kế hoạch trực tiếp có 2 lợi ích. Trước nhất, các hàng kết quả có thể được truy xuất một ít ở một thời điểm, tránh tràn bộ nhớ đối với các truy vấn trả về nhiều hàng. Thứ 2, cổng (portal) có thể vượt qua được thủ tục hiện hành (nó có thể, trong thực tế, sống tới cuối giao dịch hiện hành). Việc trả về tên cổng cho trình gọi thủ tục đưa ra một cách thức trả về một tập hợp các hàng như là kết quả.

Dữ liệu tham số được truyền sẽ được sao chép vào cổng của con trỏ, sao cho nó có thể được giải phóng trong khi con trỏ đó vẫn còn tồn tại.

Đối số

const char * name

tên của cổng (portal), hoặc NULL để cho phép hệ thống chọn tên

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

Datum * values

Mảng các giá trị tham số thực. Phải có độ dài y hệt như là số các đối số của kế hoạch.

const char * nulls

Mảng mô tả các tham số nào là null. Phải có độ dài y hệt như số các đối số của kế hoạch. n chỉ một giá trị null (khoản đầu vào trong values sẽ bị bỏ qua); một không gian chỉ một giá trị không null (nonnull) (khoản đầu vào trong values là hợp lệ).

Nếu là NULL thì SPI_cursor_open giả thiết là không tham số nào là null.

bool read_only

true cho thực thi chỉ đọc

Giá trị trả về

Con trỏ (pointer) tới cổng chứa con trỏ điểm nháy (cursor). Lưu ý là không có quy ước nào trả về lỗi; bất kỳ lỗi nào cũng sẽ được nêu qua elog.

SPI_cursor_open_with_args

Tên

SPI_cursor_open_with_args - thiết lập con trỏ bằng việc sử dụng một truy vấn và các tham số

Tóm tắt

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Mô tả

SPI_cursor_open_with_args thiết lập con trỏ (nội bộ, một cổng) mà sẽ thực thi truy vấn được chỉ định. Hầu hết các tham số có cùng ý nghĩa như các tham số tương ứng đối với SPI_prepare_cursor và SPI_cursor_open.

Đối với thực thi truy vấn 1 lần, hàm này sẽ được ưu tiên hơn SPI_prepare_cursor, theo sau là SPI_cursor_open. Nếu lệnh y hệt sẽ được thực thi với nhiều tham số khác nhau, thì hoặc phương pháp nào có thể nhanh hơn, phụ thuộc vào chi phí của việc lên kế hoạch lại so với lợi lích của các kế hoạch tùy biến.

Dữ liệu tham số được truyền vào sẽ được sao chép vào cổng con trỏ, nên nó có thể được giải phóng trong khi con trỏ vẫn còn đang tồn tại.

Đối số

const char * name

tên của cổng, hoặc NULL để cho hệ thống chọn tên

const char * command

chuỗi lệnh

int nargs

số các tham số đầu vào ($1, $2,...)

Oid * argtypes

mảng chứa các OID các dạng dữ liệu của các tham số

Datum * values

mảng các giá trị tham số thực

const char * nulls

mảng mô tả các tham số nào là null

Nếu nulls is NULL là NULL thì SPI_cursor_open_with_args giả thiết không tham số nào là null.

bool read_only

đúng cho thực thi chỉ đọc

int cursorOptions

mặt nạ bit số nguyên các lựa chọn con trỏ; zero là hành vi mặc định

Giá trị trả về

Con trỏ (pointer) tới cổng chứa con trỏ điểm nháy (cursor). Lưu ý không có qui ước trả về lỗi; bất kỳ lỗi nào cũng sẽ được nêu qua elog.

SPI-cursor_open_with_paramlist

Tên

SPI_cursor_open_with_paramlist - thiết lập con trỏ bằng việc sử dụng các tham số

Tóm tắt

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Mô tả

SPI_cursor_open_with_paramlist thiết lập con trỏ (nội bộ, một cổng) mà sẽ thực thi kế hoạch được SPI_prepare chuẩn bị. Hàm này là tương đương với SPI_cursor_open ngoại trừ thông tin về các giá trị tham số được truyền tới truy vấn được trình bày khác. Sự trình bày của ParamListInfo có thể thuận tiện cho việc truyền các giá trị có sẵn rồi ở định dạng đó. Nó cũng hỗ trợ sử dụng các tập hợp tham số động qua các hàm móc được chỉ định trong ParamListInfo.

Dữ liệu tham số được truyền vào sẽ được sao chép vào cổng con trỏ, nên nó có thể được giải phóng trong khi con trỏ điểm nháy vẫn còn tồn tại.

Đối số

const char * name

tên cho cổng, hoặc NULL để cho phép hệ thống chọn tên

SPIPlanPtr plan

kế hoạch thực thi (được SPI_prepare trả về)

ParamListInfo params

cấu trúc dữ liệu chứa các dạng và các giá trị tham số: NULL nếu không có

bool read_only

true cho thực thi chỉ đọc

Giá trị trả về

Con trỏ (pointer) tới cổng chứa con trỏ điểm nháy (cursor). Lưu ý là không có quy ước trả về lỗi; bất kỳ lỗi nào cũng sẽ được nêu qua elog.

SPI_cursor_find

Tên

SPI_cursor_find - tìm một con trỏ đang tồn tại theo tên

Tóm tắt

```
Portal SPI_cursor_find(const char * name)
```

Mô tả

SPI_cursor_find tìm một cổng đang tồn tại theo tên. Điều này trước hết là hữu dụng để giải quyết tên con trỏ được vài hàm khác trả về như là văn bản.

Đối số

const char * name

tên của cổng

Giá trị trả về

con trỏ tới cổng với tên được chỉ định, hoặc NULL nếu không có tên nào được tìm thấy

SPI_cursor_fetch

Tên

SPI_cursor_fetch - lấy vài hàng từ một con trỏ

Tóm tắt

void SPI_cursor_fetch(Portal portal, bool forward, long count)

Mô tả

SPI_cursor_fetch lấy vài hàng từ con trỏ. Điều này là tương đương với một tập con của lệnh SQL FETCH (xem SPI_scroll_cursor_fetch để có thêm chức năng).

Đối số

Portal portal

cổng có chứa con trỏ

bool forward

đúng cho lấy tiến lên, sai cho lấy lùi lại

long count

số cực đại các hàng để lấy

Giá trị trả về

SPI_processed và SPI_tuptable được thiết lập như trong SPI_execute nếu thành công.

Lưu ý

Việc lấy lùi lại có thể hỏng nếu kế hoạch của con trỏ đã không được tạo ra với lựa chọn CURSOR_OPT_SCROLL

SPI_cursor_move

Tên

SPI_cursor_move - dịch chuyển con trỏ

Tóm tắt

void SPI_cursor_move(Portal portal, bool forward, long count)

Mô tả

SPI_cursor_move bỏ qua vài số lượng hàng trong một con trỏ. Điều này là tương đương với một tập con lệnh SQL MOVE (xem SPI_scroll_cursor_move để có thêm chức năng).

Đối số

Cổng portal

cổng chứa con trỏ

bool forward

đúng cho dịch chuyển tiến lên, sai cho dịch chuyển lùi xuống

long count

số lượng cực đại các hàng phải dịch chuyển

Lưu ý

Dịch chuyển lùi xuống có thể hỏng nếu kế hoạch của con trỏ đã không được tạo ra với lựa chọn CURSOR_OPT_SCROLL.

SPI_scroll_cursor_fetch

Tên

SPI_scroll_cursor_fetch - lấy một số hàng từ con trỏ

Tóm tắt

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

Mô tả

SPI_scroll_cursor_fetch lấy vài hàng từ một con trỏ. Điều này là tương đương với lệnh SQL FETCH.

Đối số

Cổng portal

cổng chứa con trỏ

FetchDirection direction

một trong số FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE hoặc FETCH_RELATIVE

long count

số lượng các hàng để lấy cho FETCH_FORWARD hoặc FETCH_BACKWARD; số hàng tuyệt đối để lấy đối với FETCH_ABSOLUTE; hoặc số hàng tương đối cho FETCH_RELATIVE

Giá trị trả về

SPI_processed và SPI_tuptable được thiết lập trong SPI_execute nếu thành công.

Lưu ý

Xem lệnh SQL FETCH để có các chi tiết về sự diễn giải các tham số direction và count.

Các giá trị hướng khác với FETCH_FORWARD có thể hỏng nếu kế hoạch của con trỏ đã không được tạo ra với lựa chọn CURSOR_OPT_SCROLL.

SPI_scroll_cursor_move

Tên

SPI_scroll_cursor_move - dịch chuyển con trỏ

Tóm tắt

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

Mô tả

SPI_scroll_cursor_move bỏ qua vài số lượng hàng trong một con trỏ. Điều này là tương đương với lệnh SQL MOVE.

Đối số

Cổng portal

cổng chứa con trỏ

Hướng lấy direction

một trong số FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE hoặc FETCH_RELATIVE

long count

số các hàng dịch chuyển cho FETCH_FORWARD hoặc FETCH_BACKWARD; số hàng tuyệt đối để dịch chuyển cho FETCH_ABSOLUTE; hoặc số hàng tương đối để dịch chuyển cho FETCH_RELATIVE

Giá trị trả về

SPI_processed được thiết lập như trong SPI_execute nếu thành công. SPI_tuptable được thiết lập về NULL, vì không hàng nào được hàm này trả về.

Lưu ý

Xem lệnh SQL FETCH để có các chi tiết về sự diễn giải các tham số direction và count.

Các giá trị hướng khác với FETCH_FORWARD có thể hỏng nếu kế hoạch của con trỏ đã không được tạo ra với lựa chọn CURSOR_OPT_SCROLL.

SPI_cursor_close

Tên

SPI_cursor_close - đóng con trỏ

Tóm tắt

void SPI_cursor_close(Portal portal)

Mô tả

SPI_cursor_close đóng một con trỏ được tạo ra trước đó và phát hành lưu trữ cổng của nó.

Tất cả các con trỏ mở sẽ được tự động đóng lại vào cuối giao dịch. SPI_cursor_close chỉ cần được gọi nếu là mong muốn để phát hành các tài nguyên sớm hơn.

Đối số

Cổng portal

cổng chứa con trỏ

SPI_saveplan

Tên

SPI_saveplan - lưu kế hoạch

Tóm tắt

SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)

Mô tả

SPI_saveplan lưu kế hoạch được truyền (được SPI_prepare chuẩn bị) trong bộ nhớ mà sẽ không được SPI_finish và trình quản lý giao dịch giải phóng, và trả về một con trỏ cho kế hoạch được lưu. Điều này trao cho bạn khả năng sử dụng lại các kế hoạch được chuẩn bị trong các lời triệu gọi tiếp sau của thủ tục của bạn trong phiên làm việc hiện hành.

Đối số

SPIPlanPtr plan

kế hoạch sẽ được lưu

Giá trị trả về

Con trỏ tới kế hoạch được lưu; NULL nếu không thành công. Nếu có lỗi, SPI_result được thiết lập:

SPI_ERROR_ARGUMENT

nếu plan là NULL hoặc không hợp lệ

SPI_ERROR_UNCONNECTED

nếu được gọi từ một thủ tục không được kết nối

Lưu ý

Kế hoạch được truyền tới không được giải phóng, nên bạn có thể muốn làm SPI_freeplan trong nó để tránh rò rỉ bộ nhớ cho tới SPI_finish.

Nếu một trong các đối tượng (một bảng, hàm, ...) được kế hoạch chuẩn bị trước tham chiếu tới bị bỏ hoặc được định nghĩa lại, thì các thực thi trong tương lai của SPI_execute_plan có thể hỏng hoặc trả về các kết quả khác với kế hoạch ban đầu chỉ ra.

43.2. Hàm hỗ trợ giao diện

Các hàm được mô tả ở đây đưa ra một giao diện cho việc trích xuất thông tin từ các tập kết quả được SPI_execute và các hàm SPI khác trả về.

Tất cả các hàm được mô tả trong phần này có thể được cả thủ tục kết nối và không kết nối sử dụng.

SPI_fname

Tên

SPI_fname - xác định tên cột cho số cột được chỉ định

Tóm tắt

char * SPI_fname(TupleDesc rowdesc, int colnumber)

Mô tả

SPI_fname trả về một bản sao tên cột của cột được chỉ định. (Bạn có thể sử dụng pfree để phát hành bản sao tên đó khi bạn không cần nó nữa).

Đối số

TupleDesc rowdesc

mô tả hàng đầu vào

int colnumber

số cột (bắt đầu đếm từ 1)

Giá trị trả về

Tên cột; NULL nếu colnumber nằm ngoài dải. SPI_result được thiết lập về SPI_ERROR_NOATTRIBUTE khi có lỗi.

SPI_fnumber

Tên

SPI_fnumber - xác định số cột cho cột với tên được chỉ định.

Nếu colname tham chiếu tới một cột hệ thống (như, oid) thì số cột âm phù hợp sẽ được trả về. Trình gọi sẽ cẩn thận để kiểm tra giá trị trả về chính xác ngang bằng cho SPI_ERROR_NOATTRIBUTE để dò tìm lỗi; việc kiểm thử kết quả nhỏ hơn hoặc bằng 0 là không đúng trừ phi các cột hệ thống sẽ bị khước từ.

Đối số

TupleDesc rowdesc

mô tả hàng đầu vào

const char * colname

tên cột

Giá trị trả về

Số cột (bắt đầu đếm từ 1), hoặc SPI_ERROR_NOATTRIBUTE nếu cột có tên chưa được tìm thấy.

SPI_getvalue

Tên

SPI_getvalue - trả về giá trị chuỗi của cột được chỉ định

Tóm tắt

char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)

Mô tả

SPI_getvalue trả về trình bày chuỗi giá trị của cột được chỉ định.

Kết quả đó được trả về trong bộ nhớ được phân bổ bằng việc sử dụng palloc. (Bạn có thể sử dụng pfree để phát hành bộ nhớ khi bạn không cần nó nữa).

Đối số

HeapTuple row

hàng đầu vào sẽ được kiểm tra

TupleDesc rowdesc

mô tả hàng đầu vào

int colnumber

số cột (bắt đầu đếm từ 1)

Giá trị trả về

Giá trị cột, hoặc NULL nếu cột đó là null, colnumber nằm ngoài dãy (SPI_result được thiết lập về SPI_ERROR_NOATTRIBUTE), hoặc không hàm đầu ra nào sẵn sàng (SPI_result được thiết lập về SPI_ERROR_NOOUTFUNC).

SPI_getbinval

Tên

SPI_getbinval - trả về giá trị nhị phân của cột được chỉ định

Tóm tắt

Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
bool * isnull)

Mô tả

SPI_getbinval trả về giá trị cột được chỉ định ở dạng nội bộ (như dạng Datum).

Hàm này không phân bổ không gian mới cho datum. Trong trường hợp dạng dữ liệu truyền bằng tham chiếu, thì giá trị trả về sẽ là một con trỏ trong hàng được truyền.

Đối số

HeapTuple row

hàng đầu vào sẽ được kiểm tra

TupleDesc rowdesc

mô tả hàng đầu vào

int colnumber

số cột (bắt đầu đếm từ 1)

bool * isnull

cờ cho giá trị null trong cột

Giá trị trả về

Giá trị nhị phân của cột được trả về. Biến được trỏ tới bằng isnull được thiết lập về đúng (true) nếu cột đó là null, nếu không là sai (false).

SPI_result được thiết lập về SPI_ERROR_NOATTRIBUTE nếu có lỗi.

SPI_gettype

Tên

SPI_gettype - trả về tên dạng dữ liệu của cột được chỉ định

Tóm tắt

char * SPI_gettype(TupleDesc rowdesc, int colnumber)

SPI_gettype trả về bản sao tên dạng dữ liệu của cột được chỉ định. (Bạn có thể sử dụng pfree để phát hành bản sao tên đó khi bạn không cần nó nữa).

Đối số

TupleDesc rowdesc

mô tả hàng đầu vào

int colnumber

số cột (bắt đầu đếm từ 1)

Giá trị trả về

Tên dạng dữ liệu của cột được chỉ định, hoặc NULL nếu có lỗi. SPI_result được thiết lập về SPI_ERROR_NOATTRIBUTE nếu có lỗi.

SPI_gettypeid

Tên

SPI_gettypeid - trả về OID dạng dữ liệu của cột được chỉ định

Tóm tắt

Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)

Mô tả

SPI_gettypeid trả về OID dạng dữ liệu của cột được chỉ định.

Đối số

TupleDesc rowdesc

mô tả hàng đầu vào

int colnumber

số cột (bắt đầu đếm từ 1)

Giá trị trả về

OID dạng dữ liệu của cột được chỉ định hoặc InvalidOid nếu có lỗi. Nếu có lỗi, SPI_result được thiết lập về SPI_ERROR_NOATTRIBUTE.

SPI_getrelname

Tên

SPI_getrelname - trả về tên quan hệ được chỉ định

Tóm tắt

char * SPI_getrelname(Relation rel)

Mô tả

SPI_getrelname trả về bản sao tên quan hệ được chỉ định. (Bạn có thể sử dụng pfree để phát hành bản sao tên khi bạn không cần nó nữa).

Đối số

Relation rel

quan hệ đầu vào

Giá trị trả về

Tên của quan hệ được chỉ định.

SPI_getnspname

Tên

SPI_getnspname - trả về không gian tên của quan hệ được chỉ định

Tóm tắt

char * SPI_getnspname(Relation rel)

Mô tả

SPI_getnspname trả về bản sao tên của không gian tên được Relation chỉ định thuộc về. Điều này là tương đương với sơ đồ quan hệ. Bạn nên pfree giá trị trả về của hàm này khi bạn kết thúc với nó.

Đối số

Relation rel

quan hệ đầu vào

Giá trị trả về

Tên của không gian tên của quan hệ được chỉ định.

43.3. Quản lý bộ nhớ

PostgreSQL phân bổ bộ nhớ trong các *ngữ cảnh bộ nhớ*, chúng đưa ra phương pháp thuận lợi cho việc quản lý các phân bổ được làm ở nhiều chỗ khác nhau cần phải sống cho các lượng thời gian khác nhau. Việc phá hủy một ngữ cảnh đưa ra tất cả bộ nhớ đã được phân bổ trong nó. Vì thế không cần thiết lần vết các đối tượng riêng rẽ để tránh các rò rỉ bộ nhớ; thay vào đó chỉ một số ngữ cảnh nhỏ phải được quản lý. palloc và các hàm có liên quan phân bổ bộ nhớ từ ngữ cảnh "hiện hành".

SPI_connect tạo ngữ cảnh bộ nhớ mới và làm cho nó hiện hành. SPI_finish phục hồi ngữ cảnh bộ nhớ hiện hành trước đó và phá hủy ngữ cảnh được SPI_connect tạo ra. Các hành động đó đảm bảo rằng các phân bổ bộ nhớ ngắn ngủi được làm bên trong thủ tục của bạn được công bố ở lối ra của thủ tục, để tránh rò rỉ bộ nhớ.

Tuy nhiên, nếu thủ tục của bạn cần phải trả về một đối tượng trong bộ nhớ được phân bổ (như một giá trị của dạng dữ liệu đi qua tham chiếu), thì bạn không thể phân bổ bộ nhớ đó bằng việc sử dụng palloc, ít nhất không trong khi bạn được kết nối tới SPI. Nếu bạn cố làm, thì đối tượng đó sẽ bị SPI_finish bỏ phân bổ, và thủ tục của bạn sẽ không làm việc tin cậy được. Để giải quyết vấn đề này, hãy sử dụng SPI_palloc để phân bổ bộ nhớ cho đối tượng trả về của bạn. SPI_palloc phân bổ bộ nhớ theo "ngữ cảnh trình thực thi trên cao hơn", đó là, ngữ cảnh bộ nhớ mà đã hiện hành khi SPI_connect đã được gọi, nó chính xác là ngữ cảnh đúng cho giá trị được trả về từ thủ tục của bạn.

Nếu SPI_palloc được gọi khi thủ tục không được kết nối tới SPI, thì nó hành động y hệt như palloc thông thường. Trước khi thủ tục kết nối tới trình quản lý SPI, ngữ cảnh bộ nhớ hiện hành là ngữ cảnh của trình thực thi ở cao hơn, nên tất cả các phân bổ được thủ tục đó làm qua palloc hoặc bằng các hàm tiện ích SPI được làm trong ngữ cảnh này.

Khi SPI_connect được gọi, ngữ cảnh riêng tư của thủ tục, điều được SPI_connect tạo ra, được làm thành ngữ cảnh hiện hành. Tất cả các phân bổ được palloc, repalloc hoặc các hàm tiện ích SPI làm ra (ngoại trừ SPI_copytuple, SPI_returntuple, SPI_modifytuple, và SPI_palloc) sẽ được làm trong ngữ cảnh này. Khi thủ tục bỏ kết nối từ trình quản lý SPI (qua SPI_palloc) thì ngữ cảnh hiện hành được phục hồi về ngữ cảnh của trình thực thi ở cao hơn, và tất cả phân bổ được làm trong ngữ cảnh bộ nhớ thủ tục đó được giải phóng và không thể được sử dụng hơn nữa.

Tất cả các hành động được mô tả trong phần này có thể được cả 2 thủ tục có kết nối và không có kết nối sử dụng. Trong một thủ tục không có kết nối, chúng hành động y hệt như các hàm máy chủ thông thường nằm bên dưới (palloc, ...).

SPI_palloc

Tên

SPI_palloc- phân bổ bộ nhớ trong ngữ cảnh của trình thực thi ở cao hơn

Tóm tắt

```
void * SPI_palloc(Size size)
```

Mô tả

SPI_palloc phân bổ bộ nhớ trong ngữ cảnh của trình thực thi ở cao hơn.

Đối số

Kích cỡ size

kích cỡ theo byte của lưu trữ để phân bổ

Giá trị trả về

con trỏ tới không gian lưu trữ mới của kích cỡ được chỉ định

SPI_repalloc

Tên

SPI_repalloc - phân bổ lại bộ nhớ trong ngữ cảnh của trình thực thi ở cao hơn

Tóm tắt

void * SPI_repalloc(void * pointer, Size size)

Mô tả

SPI_repalloc thay đổi kích cỡ của đoạn bộ nhớ trước đó được phân bổ bằng việc sử dụng SPI_palloc. Hàm này không còn khác với repalloc bình thường nữa. Nó được giữ chỉ vì tính tương thích ngược của mã đang tồn tại.

Đối số

void * pointer

con trỏ tới lưu trữ để thay đổi đang tồn tại

Kích cỡ size

kích cỡ theo byte của lưu trữ để phân bổ

Giá trị trả về

con trỏ tới không gian lưu trữ mới của kích cỡ được chỉ định với các nội dung được sao chép từ vùng đang tồn tại

SPI_pfree

Tên

SPI_pfree - giải phóng bộ nhớ trong ngữ cảnh của trình thực thi ở cao hơn

Tóm tắt

void SPI_pfree(void * pointer)

Mô tả

SPI_pfree giải phóng bộ nhớ được phân bổ trước đó bằng việc sử dụng SPI_palloc hoặc SPI_repalloc.

Hàm này không còn khác với pfree thông thường nữa. Nó được giữ chỉ vì tính tương thích ngược của mã đang tồn tại.

Đối số

void * pointer

con trỏ tới lưu trữ đang tồn tại để giải phóng

SPI_copytuple

Tên

SPI_copytuple - làm một bản sao một hàng trong ngữ cảnh của trình thực thi ở cao hơn

Tóm tắt

HeapTuple SPI_copytuple(HeapTuple row)

Mô tả

SPI_copytuple làm một bản sao hàng trong ngữ cảnh của trình thực thi ở cao hơn. Điều này thường được sử dụng để trả về hàng được sửa đổi từ một trigger. Trong hàm được khai báo để trả về dạng tổng hợp, thay vào đó hãy sử dụng SPI_returntuple.

Đối số

HeapTuple row

hàng sẽ được sao chép

Giá trị trả về

hàng được sao chép; NULL chỉ nếu tuple là NULL

SPI_returntuple

Tên

SPI_returntuple - chuẩn bị trả về một bộ dữ liệu như một Datum

Tóm tắt

HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)

Mô tả

SPI_returntuple tạo một bản sao một hàng trong ngữ cảnh của trình thực thi ở phần cao hơn, trả nó về ở dạng của hàng dạng Datum. Con trỏ được trả về chỉ cần được chuyển đổi thành Datum qua PointerGetDatum trước khi trả về.

Lưu ý rằng điều này nên được sử dụng cho các hàm được khai báo để trả về các dạng tổng hợp. Nó không được sử dụng cho các trigger; hãy sử dụng SPI_copytuple cho việc trả về một hàng được sửa đổi trong một trigger.

Đối số

HeapTuple row

hàng sẽ được sao chép

TupleDesc rowdesc

trình mô tả cho hàng (truyền trình mô tả y hệt mỗi lẫn cho hầu hết việc giữ tạm có hiệu quả)

Giá trị trả về

HeapTupleHeader trỏ tới hàng được sao chép; NULL chỉ nếu row hoặc or là NULL

SPI_modifytuple

SPI_modifytuple - tạo một hàng bằng việc thay thế các trường được chọn của một hàng được đưa ra

Tóm tắt

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

Mô tả

SPI_modifytuple tạo một hàng mới bằng việc thay thế các giá trị mới cho các cột được chọn, sao chép các cột của hàng gốc ban đầu ở các vị trí khác. Hàng đầu vào không được sửa đổi.

Đối số

Relation rel

Chỉ được sử dụng như nguồn của trình mô tả hàng cho hàng đó. (Việc truyền một quan hệ thay vì một trình mô tả hàng là một tính năng sai).

HeapTuple row

hàng sẽ được sửa đổi

int ncols

số các số cột trong mảng colnum

int * colnum

mảng của các số các cột sẽ bị thay đổi (số cột bắt đầu ở 1)

Datum * values

các giá trị mới cho các cột được chỉ định

const char * Nulls

các giá trị mới là null, nếu bất kỳ (xem SPI_execute_plan về định dạng)

Giá trị trả về

hàng mới với các sửa đổi, được phân bổ trong ngữ cảnh của trình thực thi ở cao hơn; NULL chỉ nếu row là NULL

Nếu có lỗi, SPI_result được thiết lập như sau:

SPI_ERROR_ARGUMENT

nếu rel là NULL, hoặc nếu row là NULL, hoặc nếu ncols là nhỏ hơn hoặc bằng 0, hoặc nếu colnum là NULL, hoặc nếu values là NULL.

SPI_ERROR_NOATTRIBUTE

nếu colnum chứa một số cột không hợp lệ (nhỏ hơn hoặc bằng 0 hoặc lớn hơn số cột trong hàng).

SPI_freetuple

Tên

SPI_freetuple - giải phóng hàng được phân bổ trong ngữ cảnh của trình thực thi ở phần cao hơn

Tóm tắt

void SPI_freetuple(HeapTuple row)

Mô tả

SPI_freetuple giải phóng một hàng được phân bổ trước đó trong ngữ cảnh của trình thực thi ở cao hơn.

Hàm này không còn khác với heap_freetuple thông thường nữa. Nó được giữ chỉ vì tính tương thích ngược của mã đang tồn tại.

Đối số

HeapTuple row

hàng để giải phóng

SPI_freetuptable

Tên

SPI_freetuptable - giải phóng một tập hợp hàng được SPI_execute hoặc một hàm tương tự tạo ra

Tóm tắt

void SPI_freetuptable(SPITupleTable * tuptable)

Mô tả

SPI_freetuptable giải phóng một tập hợp hàng được một hàm thực thi lệnh SPI trước đó tạo ra, như SPI_execute. Vì thế, hàm này thường được gọi với biến toàn thể SPI_tupletable như là đối số.

Hàm này là hữu dụng nếu một thủ tục SPI cần thực thi nhiều lệnh và không muốn giữ các kết quả của các lệnh trước đó xung quanh cho tới khi kết thúc. Lưu ý rằng bất kỳ tập hợp hàng không được giải phóng nào cũng sẽ được giải phóng bằng mọi cách ở SPI_finish.

Đối số

SPITupleTable * tuptable

con trỏ tới tập hợp hàng để giải phóng

SPI_freeplan

Tên

SPI_freeplan - giải phóng một kế hoạch được lưu trước đó

Tóm tắt

int SPI_freeplan(SPIPlanPtr plan)

Mô tả

SPI_freeplan phát hành một kế hoạch thực thi lệnh được SPI_prepare trả về hoặc được SPI_saveplan lưu trước đó.

Đối số

SPIPlanPtr plan

con trỏ tới kế hoạch để giải phóng

Giá trị trả về

SPI_ERROR_ARGUMENT nếu kế hoạch là NULL hoặc không hợp lệ

43.4. Khả năng nhìn thấy những thay đổi dữ liệu

Các qui tắc sau đây điều hành khả năng nhìn thấy các thay đổi dữ liệu trong các hàm sử dụng SPI (hoặc bất kỳ hàm C nào khác):

- Trong quá trình thực thi một lệnh SQL, bất kỳ thay đổi dữ liệu nào được lệnh đó làm sẽ là nhìn thấy được đối với bản thân lệnh đó. Ví dụ, trong:
  INSERT INTO a SELECT * FROM a;
  các hàng được chèn vào sẽ nhìn thấy được đối với phần SELECT.
- Các thay đổi được một lệnh C thực hiện là nhìn thấy được đối với tất cả các lệnh được bắt đầu sau C, bất kể liệu chúng được bắt đầu trong C (trong quá trình thực thi C) hoặc sau khi C được thực hiện xong hay không.
- Các lệnh được thực thi thông qua SPI bên trong một hàm được một lệnh SQL gọi (hoặc một hàm hoặc một trigger thông thường) đi sau một hoặc các qui tắc khác ở trên phụ thuộc vào cờ đọc/ghi được truyền tới SPI. Các lệnh được thực thi trong chế độ chỉ đọc tuân theo quy tắc đầu tiên: chúng không thể thấy những thay đổi của lệnh đang gọi. Các lệnh được thực thi trong chế độ chỉ ghi tuân theo quy tắc thứ 2: chúng có thể thấy tất cả các thay đổi được làm cho tới nay.
- Tất cả các ngôn ngữ thủ tục tiêu chuẩn đã thiết lập chế độ đọc-ghi SPI phụ thuộc vào thuộc tính hay thay đổi của hàm đó. Các lệnh của các hàm STABLE và and được thực hiện ở chế độ chỉ đọc, trong khi các lệnh của các hàm VOLATILE được thực hiện ở chế độ đọc - ghi. Trong khi các tác giả các hàm C có khả năng hay thay đổi qui ước này, thì không chắc sẽ là ý tưởng tốt để làm thế.

Phần tiếp sau có một ví dụ minh hoạc ứng dụng các quy tắc đó.

43.5. Ví dụ

Phần này có một ví dụ rất đơn giản về sử dụng SPI. Thủ tục execq lấy một lệnh SQL như đối số đầu tiên của nó và một sự đếm hàng như là đối số thứ 2, thực thi lệnh bằng việc sử dụng SPI_exec và trả về số các hàng đã được lệnh đó xử lý. Bạn có thể thấy nhiều ví dụ phức tạp hơn cho SPI trong cây nguồn trong src/test/regress/regress.c và trong contrib/spi.

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int execq(text *sql, int cnt);
```

```
int
execq(text *sql, int cnt)
{
        char *command;
        int ret;
        int proc;

        /* Convert given text object to a C string */

        command = text_to_cstring(sql);

        SPI_connect();

        ret = SPI_exec(command, cnt);

        proc = SPI_processed;
        /*
        * If some rows were fetched, print them via elog(INFO).
        */
        if (ret > 0 && SPI_tuptable != NULL)
        {
                TupleDesc tupdesc = SPI_tuptable->tupdesc;
                SPITupleTable *tuptable = SPI_tuptable;
                char buf[8192];
                int i, j;

                for (j = 0; j < proc; j++)
                {
                        HeapTuple tuple = tuptable->vals[j];

                        for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                                        SPI_getvalue(tuple, tupdesc, i),
                                        (i == tupdesc->natts) ? " " : " |");
                        elog(INFO, "EXECQ: %s", buf);
                }
        }
        SPI_finish();
        pfree(command);

        return (proc);
}
```

(Hàm này sử dụng quy ước gọi phiên bản 0, làm cho ví dụ đó dễ dàng hơn để hiểu. Trong các ứng dụng thực tế bạn nên sử dụng giao diện phiên bản 1).

Đây là cách mà bạn khai báo hàm sau khi đã biên dịch nó thành một thư viện được chia sẻ (các chi tiết là trong Phần 35.9.6):

```
CREATE FUNCTION execq(text, integer) RETURNS integer
        AS 'filename'
        LANGUAGE C;
```

Đây là một phiên làm việc mẫu:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
execq
--------
     0
(1 row)
```

```
=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO: EXECQ: 0 -- inserted by execq
INFO: EXECQ: 1 -- returned by execq and inserted by upper INSERT
execq
--------
      2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
execq
--------
      1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO: EXECQ: 0
INFO: EXECQ: 1
INFO: EXECQ: 2 -- 0 + 2, only one row inserted - as specified

execq
--------
3 -- 10 is the max value only, 3 is the real number of rows
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
x
---
1 -- no rows in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO: EXECQ: 1
INSERT 0 1
=> SELECT * FROM a;
x
---
1
2 -- there was one row in a + 1
(2 rows)

-- This demonstrates the data changes visibility rule:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 2
INSERT 0 2
=> SELECT * FROM a;
x
---
1
2
```

```
2                               -- 2 rows * 1 (x in first row)
6                               -- 3 rows (2 + 1 just inserted) * 2 (x in second row)
(4 rows)                           ^^^^^^
                                   rows visible to execq() in different invocations
```